مختصر دليل لغات البرمجة

علي آل ياسين











مختصر دليل لغات البرمجة

نسخة منقحة ومُزيدة

علي آل ياسين

يتناول هذا الكتاب الأمور المتعلقة بالبرمجة بشكل مختلف قليلاً عن الأسلوب الشائع، حيث لا يشرح الكتاب إحدى لغات البرمجة بعينها، وإنما يناقش المفاهيم البرمجية المتنوعة وطريقة التفكير "كمُبرمج".

حيث يأخذك الكتاب في جولة عبر عالم البرمجة، في البداية يتحدث عن حلم البرمجة بالنسبة لكل مهتم بالبرمجيات، ثم يخوض في أساليب البرمجة المتنوعة.

بعد ذلك يدخل في جولة سريعة بين اللغات المعروفة ليعطيك لمحة عن تاريخها ومما استوحت كل لغة وفيما تستخدم عادة ونموذج من نصها البرمجي، وأخيراً يختتم بنصائح عن كيفية البداية في برمجة البرامج والتخطيط لها.

إذا كان لديك أي ملاحظات أو استفسارات يمكنك التواصل مع مؤلف الكتاب عبر:

www.alyassen.com

ali@alyassen.com

قام بالتنضيد والإخراج الفني لهذه الوثيقة **أحمد م. أبوزيد** كجزء من مشروع "**كتب عربية حرة**".

www.LibreBooks.org

وذلك باستخدام برمجيات حرة مفتوحة المصدر حصراً وبشكل كامل، شكراً لنظام أوبنتو لينُكس، المجموعة المكتبية ليبر أوفس، محرري الرسوميات جِمب وإنكسكيب، الخطوط الحرة:

Amiri, Droid Arabic Naskh, Droid Serif, Droid Sans Mono

وغيرهم من البرمجيات الحرة الرائعة.

الفهرس

تقديم

جاءتني فكرة هذا الكتاب بعد أن قرأت كتاب "بعد حروب البرمجيات" After the Software Wars للكاتب الرائع كيث كرتس Keith Curtis، وعلى الرغم من أن عملي هذا لم ولن يصل إلى مستوى عمل شخص في مستوى هذا المبرمج لا من حيث الكم ولا من حيث جودة المضمون. إلا أن هذا لم يمنعني من أن اشرع في كتيب يكون بمثابة دليل مختصر إلى لغات البرمجة متجنباً الإطالة المملة والتفاصيل التقنية الدقيقة أو التوغل في شرح الأكواد البرمجية التي من شأنها أن تبعد كل من ليس له اهتمام بالبرمجة.

عملي إذاً في هذا المختصر هو التعريف بالخطوط العريضة بأهم لغات البرمجة في عالمها الواسع. مع إضافة تعريف بأهم المصطلحات التي يجب أن تعرف كي نستطيع من خلالها قراءة تلك الخطوط العريضة.

وهذا الإصدار الثاني يتبع الأول الذي لاقى نجاحاً جيداً وأُستقبل استقبالاً طيباً من الأصدقاء والقرّاء في مجتمع لينكس العربي خاصة وفي المجتمع التقني العربي عامة. وبعد فترة من إهمال الكتاب واستثقال العودة للإضافة والتعديل كلمني الأستاذ أحمد أبوزيد لإعادة إحياء الكتاب وإظهاره بحلة جديدة وإخراج أكثر احترافية.

عليه أشكر الأستاذ أحمد وفريق العمل في موقع "كتب عربية حرة" وكل من ساعدني وأخذ بيدي سواء بعبارات شكر واستحسان أو نقد بناء واقتراح مفيد.

يوجد الكثير من المصطلحات البرمجية البحتة طبعاً لم استطع تفاديها وأيضاً لم يكن باستطاعتي إضافة تعريفها كلها خوف الإطالة ولكن قمت بإضافة جزء إضافي يعنى بشرح أهم تلك المصطلحات مع استعراض سريع لأهم نماذج البرمجة. أيضاً أرفقت في نهاية الكتاب جزء يعنى بكيفية إنشاء برنامج بالقليل من الجهد ودراسة بعض جوانب التطوير والأمور التي يجب أن تؤخذ في الحسبان في هذه العملية.

تخصصي في علوم الحاسب الآلي بالطبع لا يعطيني الخبرة الكافية لاستعراض كل لغات البرمجة، بل في حقيقة الأمر اللغات التي تعاملت معها هي فقط بيرل Perl كلغة أساسية و ++C و Java و Smalltalk و PHP وقليل جداً من Python فكان هناك حاجة كبيرة للترجمة الصرفة في أحيان كثيرة فيما يتعلق ببعض اللغات الأخرى، فأرجو الإشارة إلى مواطن الخلل في أي مكان أن وجد، كي أقوم بتصحيحها في الإصدارات القادمة.

علي آل ياسين

١ مايو ٢٠١٤ م

٢ رجب ١٤٣٥ هـ

الأحساء

تمهيد

البداية

لا أريد أن امثل دور المبرمج المحنك هنا، فأنا في الواقع لست أهلاً لذلك، ولكن بحكم تخصصي في هذا المجال واهتمامي بالجانب النظري منه ولأنني رأيت الكثير من المواضيع والأسئلة في المنتديات عن البداية في البرمجة، أحببت أن اكتب بعض الأسطر لمن لديهم اهتمام بالبرمجة ولكنهم لم يقدموا بعد.

هناك نقاط كثيرة يجب أن تؤخذ في الحسبان قبل الشروع في تعلم لغة برمجة، ولكن لاحظت أن المشكلة الأساسية عندنا هي ليست في عدم القدرة على اختيار لغة معينة بل في عدم العزم على التعلم ابتداء. من منا لا يحب أن يقال عنه مبرمج؟ خاصة أن البرمجة مثل الرياضيات يوصف أصحابها بالذكاء والعبقرية. الكثير منا يقول ويتمنى ويخطط أن يكون مبرمجاً عندما يقرأ عن المبرمجين والمخترقين وقصص نجاح البرامج والمواقع ولكن عندما يبدأ أول خطوة في الدراسة يعرف أن هناك الكثير ليتعلمه وان المادة ليست ممتعة كما كان يتصور ويبدأ بالشعور بالممل ومن ثم يترك ما شرع في قراءته من كتاب أو دورة تعليمية.

هناك حكمة يابانية تقول: «الطموح بدون عمل مجرد حلم يقظة». شخصياً كنت أعاني من هذه المشكلة وهي تكديس الكتب والدورات والمحاضرات وغيرها على أمل أني سأقرأها وأشرع في تعلمها في وقت ما! والآن لها سنوات ولم اقرأ الكثير منها! وهذه مشكلة عامة في عصر الانفجار المعلوماتي فالإنسان فعلياً يغرق في بحر من المعلومات والمصادر المتوافرة.

لذلك يجب أن ندرك أن مجرد تحميل الكتب والدورات والتنقل في المنتديات وقراءة المقارنات سواء في البرمجة أو غيرها من الفنون لن يجعل منا شيئاً! بل هو التوكل على الله والبدء والإصرار في تعلم هذه الأشياء المملة والمواد الثقيلة وإكمال الكتب والدورات إلى نهاياتها بالإضافة إلى الممارسة والتفكير هو ما يضيف إلى حصيلتنا الشيء المفيد وإلا فكل العلوم والفنون بعد أن تتوغل قليلاً وتذهب نشوة الحماس تجد نفسك في تفاصيل مملة ومعان معقدة يجب أن تتقنها وتحفظها وتمارسها كي تبدأ فعلياً عملية الانتقال من مرحلة الاطلاع والثقافة العامة إلى مرحلة الاحتراف والتخصص.

حسناً، الآن أنت عازم على المواصلة والجد في تعلم لغة برمجة معينة ولكنك لا تستطيع أن تختار أي لغة تناسبك؟ هذه المشكلة الثانية ونرى الكثير من المواضيع في المنتديات على هذه الشاكلة أي لغة برمجة اختار؟ ما هي أفضل لغة برمجة؟ ما هي أقوى لغة برمجة؟ أيهما الأقوى بيرل أو بايثون؟ جافا أم سي شارب؟

في حقيقة الأمر لا يوجد شيء اسمه اللغة الأقوى وكل من يقول هناك شيء من هذا القبيل فهو إما متعصب أو واهم، فكل لغة قوية في جانب معين وعادة ما يكون هو الجانب الذي أنشئت من أجله أساساً. طبعاً يوجد لغات تصلح لكل شيء تقريباً ولكن يبقى أنها لا تقدم كفاءة وإنتاجية عالية في كل شيء، فمثلاً لغة مخصصة للويب مثل PHP وإن كان هناك إمكانية عمل برامج سطح مكتب بها فهي لن تكون بقوة وسهولة وإنتاجية لغات البرمجة المخصصة لهذا الجانب والعكس صحيح.

على هذا فسيكون الجواب على سؤالك عدة أسئلة! ما هي احتياجاتك؟ ماذا تريد أن تبرمج؟ هل تريد تعلم البرمجة لنفسك أم لسوق العمل؟ هل المشروع يحتاج إلى السرعة في الأداء أم أن السهولة في التطوير أهم؟ ما طبيعة البيانات المراد التعامل معها هل هي مستندات نصية أم صور أم إحصائيات؟ إلخ، سترى أنه بالإجابة على كل هذه الأسئلة ستختلف اللغة المطلوبة لذلك تجد أن أغلب المبرمجين يعرفون أكثر من لغة وذلك لأنهم يختارون الأداة المناسبة لكل مهمة. أيضاً هذا الكتاب ربما يكون جيداً في اختيار لغة برمجتك الأولى فهو يعرفك بالأهداف الرئيسية التي أُنشئت من أجلها لغات البرمجة المختلفة وأهم التطبيقات التي تستخدم فيها هذه اللغات.

والنقطة الجديرة بالذكر هنا أيضاً هي اطلاعك على أمثلة من طريقة كتابة الأكواد في لغات مختلفة باعتقادي الشخصي سيؤثر على اهتمامك واستمتاعك بلغة البرمجة التي تختارها فهناك الكثير من المبرمجين يستخدمون لغات معينة فقط لأنهم يحبون أسلوبها في الكتابة وأيضاً طريقة معالجتها للمشاكل البرمجية. مثلاً إذاً كنت تحب أن يكون كل شيء واضحاً في أسماء المتغيرات والكلمات المفتاحية وغيرها وتكره استخدام الكثير من الاختصارات والرموز كالأقواس الكثيرة في Lisp والأقواس المعقوفة وما إلى ذلك فذلك بالطبع سيؤثر على اختيارك.

وبمناسبة الكلام عن الأقواس فبعض المبرمجين يصف كثرة استخدام الأقواس في Lisp بجهنم الأقواس! ولكن يقول الآخر «عندما وصلت إلى مرحلة التنوير ارتفعت الأقواس!» هذه الأمور النفسية والذوقية ربما يستصغرها البعض لكن هذه الأمور فطرية ولا يمكن تجاهلها.

أتذكر أني كنت في أشد الشوق لقراءة أحد الكتب لما سمعت عنه وتلمست من مؤلفه وكان الكتاب صعب المنال لندرته وكنت أتوقع أني لو تحصلت عليه سأقرأه في جلسة ولكن عندما حصلت عليه لم استطع أن اقرأ إلا مقالة واحدة لان الورق كان رديئاً والخط كان صغيراً وضعيفاً بشكل أفسد علي متعة القراءة بشكل كامل.

الشيء الأهم هو أن تعلم أن:

- تعلمك لأي لغة برمجة سيفيدك كثيراً حتى لو انتقلت إلى لغة أخرى.
- لغات البرمجة تشترك في مفاهيم ومبادئ أساسية أن فهمتها سهل عليك الانتقال حسب الحاجة والرغبة إلى لغات أخرى.

ولهذا كما أشرنا نرى أن أغلب المبرمجين يعرفون أكثر من لغة ويسهل عليهم التأقلم مع أي لغة جديدة يفرضها عليهم سوق العمل، وكمثال في بيرل لم أكن أعرف كيف أقوم بعمل برنامج ذو واجهة رسومية لأنني كنت معتاداً على عمل برامجي على الويب أو سطر الأوامر ولكن بعد تعلمي مكتبة Tk وفهمي لكيفية عمل الواجهات الرسومية استطعت بكل بساطة ويسر أن أنقل برنامجي إلى مكتبة wxPerl.

لذلك شخصياً أرى الأهمية المطلقة التي يغفل عنها الكثيرون للجانب النظري والمنطقي للبرمجة عوضاً عن التركيز على حفظ الدوال والأكواد فهذه أشياء بسيطة يمكن الرجوع إليها بسرعة من خلال كتيب الدليل والبحث في الإنترنت، فالأفضل أن نعرف كيف يعمل البرنامج لا كيف يُكتب البرنامج.

شيء آخر يجب أن نشير إليه ألا وهو عدم الاهتمام كثيراً بما يشاع عن أفضلية طريقة معينة في البرمجة، مثلاً هناك إشارة دائمة من خلال دراستي في الجامعة إلى أفضلية البرمجة الكائنية ولكن يجب أن ندرك أن البرمجة الكائنية ليست دائماً هي الحل الأفضل للمشكلة، فقد توجد مشاكل وبرامج استخدام البرمجة الكائنية فيها مجرد تطويل وتعقيد للحل وكما قالوا: «في البرمجة الكائنية إذاً أردت موزة ستتحصل على غوريلا يمسك بموزة مع الغابة كلها»، ولكن طبعاً في برامج ونواحي أخرى سيكون من الصعب البرمجة باستخدام البرمجة الإجرائية بدلاً من البرمجة الكائنية. لذلك دائماً أحب أن أشير إلى أهمية الجانب النظري في البرمجة وهو فهم طرق البرمجة ومبادئها ومن ثم اختيار الطريقة الأفضل.

ختاماً أشير إلى أن الإنسان لا يستمر على شيء إلا إذاً كان ممتعاً والبرمجة ممتعة إذا كانت تحل مشكلة نحتاج لحلها وإلا ستكون البرمجة مملة ورتيبة وعادة ما نترك المشروع قبل إتمامه. على سبيل المثال لينوس تورفالدز Linus Torvalds (مبرمج نواة لينكس) لم يشرع في برمجة نظامه إلا من مشكلة عاناها مع نظم التشغيل الموجودة في وقته وغيره من الأمثلة كثيرة وهنا نصل إلى نقطة أخرى وهي أن البرامج العملاقة سيكون من الصعب جداً على شخص واحد فقط تطويرها وكتابتها وهنا تظهر أهمية العمل الجماعي والاحتكاك بالمبرمجين الآخرين والمشاركة في المحافل البرمجية ودراسة أكواد الغير للعمل من حيث انتهى الآخرون وليس تضييع الوقت في إعادة اختراع العجلة وتشتيت الجهود.

النقطة الأخيرة التي أحب أن اختم بها هي مسألة الدراسة البرمجية، ففي الجامعة نمل ونتضجر من الأشياء النظرية التي نتعلمها ونريد تعلم الأشياء "الكوول" على ما يقال! ولكن لاحقاً ندرك أن هذه الأشياء النظرية المملة هي أهم ما تعلمناه ونرى لاحقاً مكانها في عالم البرمجة، كمثال كان هناك شخص دائماً يتضجر من البرمجة لسطر الأوامر ويقول أن هذا شيء قديم أكل الدهر عليه وشرب وأنه علينا أن نتعلم البرمجة الحديثة فما نفع برامج سطر الأوامر هذه الأيام.

ولكن في حقيقة الأمر الجامعة والكتب بشكل عام تعلمك البرمجة لسطر الأوامر لأنها تركز على المنطق في حل المشاكل البرمجية وما سطر الأوامر إلا مجرد أداة لاستلام المتغيرات والتفاعل مع المستخدم. لاحقاً أدركنا أن الواجهات الرسومية وبرامج الويب والآن برامج الموبايل مجرد قشور لا تنفع إذاً لم يكن خلفها أكواد برمجية سحرية تعمل بصمت خلف الستار، وأن الذي تعلم تلك الأكواد والمبادئ وتمكن منها لم يجد صعوبة في أن يغلفها بواجهات حديثة مثل QT أو GTK.

الفصل الأول

نماذج البرمجة والمصطلحات

في البدء كان الصفر والواحد. ثم جاءت اللغات التجميعية، إلى هنا ونحن نتكلم عن اللغات منخفضة المستوى أي أنها ذات ارتباط شديد بالعتاد Hardware، وإن كنا مع اللغات التجميعية بدأنا الماكروز Macros وبعض الخصائص التي سنراها لاحقاً في اللغات الإجرائية بشكل أكثر تقدماً. ثم بزغ الفجر الجديد مع اللغات عالية المستوى وظهرت البرمجة الإجرائية لتعلن عن ظهور عصر الاهتمام بالمشكلة البرمجية!

فقبل أن نتكلم عن لغات البرمجة يجب أن نسلط الضوء على أهم نماذج البرمجة وأهم المصطلحات المرتبطة بها ليسهل علينا لاحقاً فهم لغات البرمجة من خلال قراءة أسطرها العريضة. وفي نهاية الفصل سنتعرض لمصطلحات متنوعة.

نماذج البرمجة Programing Paradigms

يجب أن ننتبه إلى أنه لا يوجد أي تعريف رسمي لأي نموذج من النماذج الآتية، ونقاط الاشتراك كثيرة فالحدود هنا ليست فواصل لا يمكن تجاوزها بل هي أشبه ما تكون بالمذاهب الأدبية في هذا الجانب فالبرمجة الشيئية في النهاية هي برمجة إجرائية بطبيعتها. ولكن ما يعنينا في حقيقة الأمر هنا هو النقاط التي يركز عليها كل نموذج من هذه النماذج حيث عليه تتغير طريقة تفكير المبرمج في نظرته وتحليله وطريقة مباشرته للمشاكل البرمجية. وهذه النماذج القليلة المذكورة هنا هي المهمة حيث يوجد العديد والعديد من النماذج الأقل انتشاراً كما هنالك الأعداد المهولة من لغات البرمجة.

البرمجة الأمرية (الإلزامية) Imperative

تصف الحوسبة من ناحية الجمل التي تغير حالة State البرنامج. وتندرج تحتها العديد من النماذج كما سيأتي.

أولاً: البرمجة غير المنظمة Non-Structured Programing

قبل البدء في الكلام عن النماذج المرتبة (الإجرائية، الكائنية) يجب أن نذكر بأنه في البداية كان هناك نموذج البرمجة غير المرتبة ويوجد لغات عالية ومنخفضة المستوى تستخدم هذا النموذج منها:

- MSX Basic .
- GWBasic.
- Focal Joss .
- Mumps .
- Telcomp .
- COBOL.
- Machine Level Code .
- Assembly Debuggers .
- وبعض نظم الأسمبلي وبعض لغات السكربتنج مثل MS-Dos Batch file Lang.

البرمجة غير المنظمة تلاقي نقداً عنيفاً لأنها تنتج أكواد صعبة القراءة أو ما يعرف "بأكواد الإسباجتي"! لهذا لا تعتبر في بعض الأوقات خياراً مناسباً للمشاريع المهمة ولكن في الجهة المقابلة يمدحها البعض للحرية التي توفرها للمبرمج ويشبهونها بطريقة كتابة موتسارت للمقطوعات الموسيقية.

البرامج التي تكتب بهذه الطريقة عادة ما تتكون من مجموعة أوامر أو جمل متتالية (غالباً كل جملة في سطر مستقل). أما الأسطر فتكون مرقمة أو معنونة "Labeled" ومن خلال هذه الآلية يمكن للبرنامج أن يقفز إلى أي سطر برمجي.

البرمجة غير المنظمة توفر أساسيات آليات التحكم بسير البرنامج وتوفر أيضاً ال Subroutines وسنتكلم عن هذه الأشياء بقليل من التفصيل في البرمجة الإجرائية والوظيفية. يبقى أن نعرف أن البرمجة غير المرتبة تتيح لنا أنواعاً مختلفة من البيانات الأولية مثل الأعداد والنصوص والقوائم.

ثانياً: البرمجة المنظمة Structured Programing

البرمجة المنظمة كالبرمجة غير المنظمة تعتبر إحدى شعب البرمجة الأمرية (إحدى أهم وأكبر نماذج البرمجة). واشتهرت البرمجة المنظمة بإزالتها للجملة الشهيرة GOTO أو الحد من استخدامها. وهناك ثلاث منهجيات مشهورة ومتداولة لكتابة البرامج المنظمة:

١. طريقة ادسجار دايجسترا Edsger Dijkstra حيث هيكل البرنامج مكون من مجموعة هياكل جزئية. بهذه الطريقة يمكن فهم البرنامج من خلال فهم كل جزء لوحده وبه نتحصل على فصل وعزل للمهمات المختلفة.

٢. طريقة أخرى مشتقة من الطريقة الأولى حيث يتم تقسيم البرنامج إلى برامج جزئية مع وجود مدخل واحد فقط للبرنامج ولكن تعارض وبقوة مبدأ المخرج الموحد.

٣. طريقة جاكسون Jackson للبرمجة المنظمة والتي تعتمد على محاذاة البيانات المنظمة مع أجزاء البرنامج المنظمة.

يوجد على الأقل ثلاث طرق أساسية لتصميم برامج البيانات المنظمة مقترحة بأسماء أصحابها:

١. J. Dominique Warner .

٢. Ken Orr .

٣. Michael Jackson .

من المنظور الدوني Low Level يمكن أن نرى البرامج المنظمة على أنها مكونة من آليات تحكم مسار البرنامج تنقسم إلى ثلاث أنواع:

- Sequence: ونقصد بها تنفيذ أوامر بترتيب تسلسلي منظم.
- Selection: ونقصد بها تنفيذ مجموعة أوامر اعتماداً على حالة البرنامج وذلك عادة من خلال الكلمات المفتاحية مثل case ... switch ... else ... then ... if.
- Repetition: ونقصد بها تنفيذ أوامر معينة وتكرار عملية التنفيذ إلى أن يصل البرنامج إلى حالة معينة أو تنفيذ مجموعة أوامر على كل عنصر من عناصر مجموعة ما. وذلك يتم عادة من خلال الكلمات المفتاحية: do ... for ... repeat ... while ... white ... until .

ربما يمر علينا مصطلح Block-Structured وهو صفة للغات البرمجة التي توفر في طريقة كتابتها الكلمات المفتاحية مع الأقواس التي تضم الأوامر المجزأة. أما النوع الثاني هو Comb-Structured وهي اللغات التي توفر آلية الكلمات المفتاحية المتسلسلة التي تحتوي بداخلها الأوامر المجزأة. مثال الأخير في لغة Ada الـ Block مكون من أربعة أجزاء DECLARE, BEGIN, EXCEPTION, END.

البرمجة الإجرائية Procedural

تعتمد على عملية توفير الخطوات اللازمة كي يصل البرنامج إلى الحالة (State) المطلوبة. وهي من اقدم النماذج وأكثرها انتشاراً وقريبة جداً من الطريقة البديهية في التفكير. وسنستعرض بعض أهم المفاهيم التي تتعامل معها باختصار لأن أغلب اللغات الحديثة تحتوي على هذه المفاهيم جلها أو بعضها.

١. المتغيرات Variables

في البرمجة كما هو الحال في الرياضيات هناك حاجة ماسة للتعامل مع المتغيرات ولا يخلو برنامج ما من متغير إلا في حالات نادرة جداً. أبسط أنواع المتغيرات هو الذي يمكنه أن يحمل قيمة واحدة فقط. في هذه الحالة "ص" متغير يحمل قيمة معينة Value ولكن هذه القيمة متغيرة فهي ليست ثابتة طالما أن البرنامج في طور التنفيذ ففي أي لحظة ممكن أن تتغير هذه القيمة. كمثال بسيط لنقل أن لدينا متغير باسم Total وقد نبدأ البرنامج وقيمته الفعلية "صفر" ولكن مع استمرار تنفيذ البرنامج ربما يتم إسناد قيمة جديدة لهذا المتغير فتكون قيمته ١٠٠ مثلاً.

ويوجد هناك نوع مختلف كلياً للمتغيرات يعرف باسم Static Variable أو Constant أو غير ذلك ومهما اختلف المصطلح فالمعنى أن هذا المتغير لا يمكن تغيير قيمته بعد أن نسند له القيمة الأولية، هذه المتغيرات قد تبدو عبثية فهي تخالف المغزى المتعارف لإنشاء المتغيرات ولكن في الحقيقة لها أغراض أخرى مفيدة منها كمثال عندما نريد أن ننشئ متغيرات لحقائق ثابتة مثلاً متغير يحمل قيمة Pi.

بعض لغات البرمجة مثل جافا تستوجب تحديد نوع المتغير فإذا كان المتغير من نوع "حرف" مثلاً فلا يمكن إسناد قيم رقمية للمتغير ولكن في لغات البرمجة الديناميكية مثل بيرل و PHP لا يوجد مثل هذا القيد فالمتغير "ص" مثلاً يمكن أن يحمل أي قيمة رقمية، نصية...إلخ.

مثال:

```
my $number = 1;
```

في هذا المثال أعلنا عن المتغير وفي نفس الوقت أسندنا له قيمة وهي "١" كان بالإمكانيات أن نعلن عن المتغير فقط من دون إسناد أي قيمة هكذا:

```
my $number;
```

مثال آخر:

```
my $name = "Ali";
my $name = "Tomy";
```

هذا المثال المتغير كان يحمل قيمة علي ولكن تم تغيير القيمة إلى "تومي" وسيظل المتغير يحمل اسم تومي إلى نهاية البرنامج إذا لم يتم إعادة تغيير هذه القيمة.أيضاً في المثال السابق name يعرف باسم variable name أو variable identifier في بعض لغات البرمجة وذلك يعني أن هذا معرف المتغير أما Ali أو Tomy فهي قيم المتغير Value.

٢. القوائم أو المجموعات Arrays

طيب، ماذا لو كنا نريد أن ننشئ متغيراً ولكن يحمل عدة قيم. مثلاً أسماء أصدقاء البيئة. يمكننا أن نستخدم القوائم وهي متغير ولكن يحمل عدة قيم بداخله وذلك بحسب ترتيب رقمي فالقائمة تبدأ من "صفر" للعنصر الأول و"١" للعنصر الثاني... وهكذا.

مرة أخرى هناك لغات برمجة تطلب تحديد نوع العناصر الموجودة في القائمة ولغات لا تطلب، مثالنا هنا قائمة بأسماء طلاب:

```
my @students = qw\Ali Yasser Salman\;
```

الآن المتغير students يحمل ثلاث قيم: علي وياسر وسلمان ويمكن الوصول إلى كل قيمة بتحديد رقم القيمة:

```
print $students[0];
```

هذا الأمر مثلاً سيطبع لنا القيمة ذات الترتيب "صفر" وهي بعبارة أخرى أول قيمة موجودة في القائمة وفي مثالنا هي Ali. طبعاً هناك الكثير من العمليات التي يمكننا أن نجريها على هكذا قائمة مثلاً إضافة عنصر أو إزالته أو إعادة ترتيب العناصر أبجدياً وغير ذلك مما لا يسع المجال أن نشير إليه في هذا المختصر.

٣. القواميس Hash

يوجد نوع آخر من المتغيرات كثيرة التداول يشبه القوائم إلى حد كبير اسمه الهاش وله أسماء أخرى ما يعنينا هنا أن القيم تمثل على شكل مفتاح key وقيمة value. بعبارة أخرى كنا في القوائم نشير إلى القيم باستخدام مفاتيح رقمية، ولكن في الهاش يمكننا نحن أن نحدد المفاتيح والتي يجب أن تكون مميزة غير مكررة وبذلك نستطيع تمثيل قواعد بيانات كاملة مثل القواميس ودليل الهاتف وسجل الطلاب ودرجاتهم...إلخ.

مثال:

```
my %employees = qw/
05650005 Ali
05900099 Yasser
08999279 Basem /;
```

في هذا المثال قمنا بإنشاء هاش يحمل أرقام الموظفين ولأن أرقام الجوالات غالباً مميزة وغير مكررة قمنا بجعلها مفاتيح للوصول إلى اسم الموظف المطلوب.

مرة أخرى يمكننا التعديل على كل عنصر في الهاش لوحده أو نقوم بعملية على جميع العناصر في وقت واحد إلى غير ذلك من العمليات التي سيطول المقام في شرحها وقد تختلف من حيث الكتابة من لغة إلى أخرى، ولكن هذا مثال بسيط لكود يطبع كل القيم الموجودة في الهاش وسنتكلم عن دوائر التكرار بعد قليل.

```
foreach $phoneNumber(keys %hash){
print "$phoneNumber = $hash{$phoneNumber}\n"}
```

٤. القيمة الخالية Null value

وهي القيمة غير المُعرفة ولا تعني الصفر كما قد يُتوهم. مثلاً الإعلان عن متغير بدون إسناد قيمة سيحتوي على قيمة غير مُعرفة، ومن الأشياء الأساسية في اختبار البرامج البحث عن القيم غير المعرفة ويمكن التحقق بأن المتغير يحمل قيمة بآليات مختلفة منها الدالة defined في بيرل ومثالها:

```
my ;$number
if (defined($number))
{ print $number }
else
{print "undefined value"}
```

تجدر الإشارة إلى أن بعض لغات البرمجة عندما لا تسند قيمة إلى المتغير بدلاً من أن تضع القيمة غير المعرفة تقوم بتزويد قيمة افتراضية بحسب نوع المتغير فيجب الانتباه إلى هذه النقطة.

٥. مسار البرنامج Workflow

عندما نكتب برنامجاً سيكون علينا في أغلب الأحيان التحكم في سير البرنامج اعتماداً على المتغيرات وطلبات المستخدم، فقط في البرامج البسيطة لا نحتاج إلى تغيير مسار البرنامج. في هذه الحالة سيقوم المفسر بتنفيذ الأوامر من البداية إلى النهاية وينتهي البرنامج بشكل تسلسلي Sequenced، مثال:

```
my $number = 1;
my $number2 = 2;
print $number+number2;
```

في هذا المثال سيقوم البرنامج بإنشاء متغير ويسند له قيمة ومتغير ثان ويسند له قيمة أخرى وفي النهاية يجمع القيمتين ويطبع الناتج على الشاشة.

نلاحظ عدم وجود أي تشعب في مسار البرنامج. ولكن هذا كما قلنا للبرامج البسيطة ولكن في أغلب الأحيان سنحتاج إلى أدوات للتحكم بسير العمل Control Structures ومرة أخرى كل لغة برمجة توفر آليات مختلفة ولكن سنأخذ بعض الأمثلة الدارجة.

الجملة الشرطية IF

باستخدام الكلمة المفتاحية if والتي تعني "إذا" نستطيع تنفيذ جزء معين من البرنامج بحسب نتيجة الشرط. مثال:

```
if ($user eq "Ali")
{print "access granted"}
```

هذا مثال بسيط لدينا عبارة تأكيد دخول ولكن لا نريد أن نطبعها لكل مستخدم يدخل البرنامج، فقط في حالة تحقق الشرط نطبع العبارة. الشرط هو ما بين القوسين الذين يليان كلمة if، وفي هذه الحالة الشرط هو أن يكون المتغير يحمل قيمة نصية هي Ali. جميل؛ الآن نستطيع أن ننفذ أمراً آخر إذاً كان الشرط غير متحققاً. وذلك يتم عن طريق استخدام كلمة else.

```
else {print "I don't know you"}
```

الآن البرنامج عندما يصل إلى الجملة الشرطية ويرى أن الشرط غير متحقق سينفذ ما هو موجود في else وفي مثالنا يطبع جملة "I don't know you".

يمكننا أيضاً إضافة المزيد من الشروط لنفس الجملة الشرطية باستخدام elsif (حرف e محذوف عمداً في لغة البرمجة بيرل).

```
elsif($user eq "Fatima")
{print " Hi Fatima"}
```

الآن أصبح لدينا برنامج بسيط له عدة مسارات فهو يقوم بالتحقق من الشرط الأول هل قيمة المستخدم Ali؟ وإذا لم يكن متحققاً سيذهب إلى الشرط الثاني ليتأكد هل قيمة المستخدم Fatima؟ فإذا كان الشرط متحققاً سيطبع الجملة المناسبة وإلا سيذهب إلى else ويطبع الجملة أنا لا أعرفك.

الجملة الشرطية Unless

الجملة الشرطية unless تعمل بعكس الجملة الشرطية if تماماً. فكأننا نقول إذاً كان الشرط غير متحقق افعل كذا وكذا. بعكس ما كنا نفعل مع if حيث كنا نقول إذاً كان الشرط متحققاً فافعل كذا وكذا.

مثال:

```
unless($a == 10)
{ print "the number isn't 10" }
```

في هذا المثال سيقوم البرنامج بطباعة جملة "هذا الرقم ليس ١٠" ما دام أن الشرط في الأعلى غير متحقق وبالطبع كنا نستطيع أن نكتب هذا بطريقة if ولكن سيكون الشرط بالنفي وليس بالإيجاب.

هكذا:

```
if($a! 10)
{print "the number isn't 10";}
```

جملة Given-When

في بعض الأحيان بدلاً من الإكثار من الجمل الشرطية if else يمكننا أن نحدد مسار البرنامج بآلية أسهل وأجمل في الكتابة حيث سنقوم بتحليل قيمة المتغير وبناءً على قيمته سنتخذ الإجراء المناسب:

```
given($name){
when("Ali"){say "welcome $name you are the 1st"}
when("Saleh"){say "welcome $name you are the 2nd"}
when("Hashem"){say "welcome $name you are the 3rd"}
default{say " I don't know you " }}
```

نلاحظ هنا كيف أن البرنامج سيختبر قيمة المتغير name وعليه سيتخذ الإجراء المناسب فنحن هنا باستخدام كلمة when سنحدد القيم المتوقعة مثلاً أن يكون الاسم علي أو صالح أو هاشم ولا ننسى أن نمرر قيمة افتراضية في حال كانت قيمة المتغير لا تتطابق مع أي من القيم التي توقعناها ف default هنا تعمل عمل else وهي الحالة العامة أو الافتراضية أن شئت.

جملة Switch case

المثال أعلاه على طريقة بيرل ولكن في بعض اللغات الأخرى طريقة الكتابة تختلف وإن كان المفهوم واحداً فبدلاً من given when نستخدم switch case حيث switch تأخذ

المتغير المراد اختباره و case تأخذ القيم المتوقعة، مثال:

```
use Switch;
switch ($name) {
case "Ali" {say "welcome $name you are the 1st"}
case "Saleh" {say "welcome $name you are the 2nd"}
case "Hashem" {say "welcome $name you are the 3rd"}
else{say "I don’t know you"}}
Expression Modifiers
```

في اللغة هناك تراكيب مختلفة للجملة مثلاً "اسمي علي" أو "علي هو اسمي" تركيبان مختلفان لنفس المعنى. في اللغات البشرية تتاح هذه الإمكانية لتفيد معنى التأكيد على جزء معين من الجملة.

مثلاً علي وخالد ذهبا لحفل التخرج. ماذا لو كنا نريد التأكيد على الحفل وليس على من ذهب للحفل نستطيع أن نقول لحفل التخرج ذهب علي وخالد! هنا سنركز اهتمام السامع إلى كلمة حفل التخرج. معدلات التعابير Expression Modifiers تعمل بمثل هذه الآلية فهي تعكس ترتيب الجمل الشرطية ودوائر التكرار وهذه أمثلة نلاحظ كيف أن الشرط يقع في نهاية الجملة بعد الأمر الذي سينفذ في حال كان الشرط متحققاً.

```
$bool = 1;
$a = 5;
say "I am in love" if $bool;
say "false" unless $bool;
print "Perl Programming…\n" , ($a++) while $a < 9;
```

٦. دوائر التكرار

في بعض الأحيان نحتاج أن نكرر تنفيذ أوامر معينة، وهنا يأتي دور دوائر التكرار، عادة ما يرافق كل دائرة تكرار عداد لديه قيمة صغرى وقيمة كبرى لكي تنتهي عملية التكرار عند الوصول للقيمة الكبرى وبذلك نضمن عدم استمرار التكرار إلى ما لا نهاية مما قد يؤدي إلى انهيار البرنامج وهذا ما يعرف باسم Infinite Loop.

لنبدأ بال for loop مثالها على طريقة لغة C:

```
for($a=0;$a<10;$a++)
{print "loop $a \n"}
```

نلاحظ أن بعد كلمة for يوجد قيمة صغرى لبداية العداد ومن ثم الشرط الذي ستتحقق منه الدائرة عند تنفيذ كل دورة وهو هنا أن يكون المتغير أقل من ١٠ وفي النهاية سنزيد قيمة العداد واحدا +١ مع انتهاء كل دورة... وبذلك ستنفذ الأوامر التي بين الأقواس المعقوفة عشر مرات لنحصل على هذا الناتج:

```
loop 0
loop 1
loop 2
loop 3
loop 4
loop 5
loop 6
loop 7
loop 8
loop 9
```

أيضاً في بيرل يمكننا أن نمرر قائمة بين قوسي ال for مثلاً من ١ إلى ١٠ حيث ستكون القيمة الصغرى ١ والقيمة الكبرى ١٠.

```
for (1…10)
{print "looping 10 times\n"}
```

هنا سنطبع الجملة التي بين الأقواس المعقوفة ١٠ مرات.

جملة Foreach

طريقة عملها مشابهة لجملة for وتختلف قليلاً بين كل لغة وأخرى ولكن في بيرل عادة ما نستخدمها لمعالجة عناصر قائمة، مثلاً:

```
my @list = qw/ Ali Sara Yosra Mamdoh/;
foreach(@list){
print $_, " You are a member of the family!
"; }
```

في هذا المثال نلاحظ أن لكل عنصر من عناصر القائمة سيُطبع اسمه ومن ثم طباعة الجملة You are a member of the family! أمامه. فتكون النتيجة:

```
Ali You are a member of the family!
Sara You are a member of the family!
Yosra You are a member of the family!
Mamdoh You are a member of the family!
```

جملة While

دائرة التكرار while تقوم بعملية مشابهة أيضاً فهي تقوم بتكرار تنفيذ أوامر معينة عدد مرات محددة ما دام الشرط متحققاً.

مثال:

```
$a = 10;
while ($a! 0)
{print $a--;}
```

في هذا المثال سنتأكد من الشرط هل المتغير لا يساوي صفراً؟ إذاً كانت الإجابة نعم سنُطبع قيمة المتغير مع إنقاصها واحداً أي "-١". فإذا كان الشرط متحققاً تنتهي الدائرة مع وصول المتغير للقيمة "صفر"، في مثالنا سيقوم بالعد تنازلياً من ١٠ إلى ١.

جملة Do while

في بعض الأحيان نريد أن نقوم بعملية التكرار مرة واحدة على الأقل بغض النظر عن كون الشرط متحققاً أو لا، هنا نستخدم do while.

```
my $a = 10;
do {
print "this is a do while loop";
$a++;
} while ($a<10);
```

نلاحظ هنا أن الأوامر الموجودة داخل الأقواس المعقوفة ستنفذ على الأقل مرة واحدة حتى لو كان الشرط الموجود في while غير متحقق. في مثالنا قيمة المتغير ١٠ والشرط هو أن نقوم بتنفيذ الأوامر إذاً كان العدد أصغر من عشرة لهذا الشرط غير متحقق ولكن لأننا استخدمنا do while فسيتم تنفيذ الأوامر مرة واحدة ومن ثم تقييم الشرط وبما أنه غير متحقق سنخرج من التكرار.

جملة Until

تقوم until بعملية عكسية لما تقوم به while فهي تبدأ بشرط غير متحقق بدلاً من الشرط المتحقق في while وستنتهي الدائرة عندما يصبح الشرط متحققاً بخلاف while التي تنتهي عندما يصبح الشرط غير متحقق، مثال:

```
my $control = 10;
until ($control == 20)
{
print "going up until 19";
$control++; }
```

في هذا المثال سنقوم بالصعود بداية من ١٠ وانتهاءً برقم ١٩ حيث سيكون الشرط متحققاً عند وصول المتغير إلى قيمة ٢٠ وبذلك ينتهي التكرار.

٦. التعابير النمطية Regular Expressions

أو اختصاراً Regex هي تعابير نمطية توفرها لغات البرمجة للتعامل مع النصوص والمدخلات بالإضافة إلى استخدامها في أدوات وأوامر إدارة النظم بكثرة. طبعاً هذه التعابير عالم في حد ذاتها ولها كتب متخصصة بها ولكن نختصر فنقول أنه هناك رموز توفرها اللغة (أيضاً تختلف من لغة إلى أخرى) لكي تسهل عملية التعامل مع النصوص كعملية اجتزاء جزء معين أو البحث عن نص بهيئة معينة. وهذه بعض الأمثلة لتقريب الفكرة فقط.

في المثال التالي سيقوم البرنامج بتحويل النص الموجود داخل المتغير string إلى حروف انجليزية كبيرة.

```
$string =~ tr/[a-z]/[A-Z]/;
```

الآن قمنا بعكس العملية فهنا سنقوم بتحويل النص إلى حروف صغيرة.

```
$string =~ tr/[A-Z]/[a-z]/;
```

هذا المثال يقوم البرنامج بالبحث في المتغير فإذا وجد "علي سالم" سيقوم بتحويلها إلى "علي إبراهيم" بالضبط كما نقوم بعمل Find and Replace في البرامج المكتبية.

```
$string =~ s/Ali Salem/Ali Ibrahim/;
```

الخلاصة كما رأينا بعض الأمثلة بلغة البرمجة بيرل يمكن إجراء عمليات كثيرة على النصوص من خلال ما يعرف بالتعابير النمطية والتي تكتب "غالباً" في بيرل بين // حيث بينها نضع التعابير النمطية وقبلها وبعدها يمكن إضافة معاملات مختلفة مثل معامل الاستبدال s كما رأينا في المثال الأخير.

البرمجة المعتمدة على الأحداث Event-Driven

في هذه الطريقة من البرمجة يحدد مسار البرنامج من خلال الحوادث Events، مثلاً: الحساسات أو من خلال المستخدم كالنقر على الفأرة أو الرسائل من برامج أخرى أو من خلال Threads. عادة نتعامل مع هذه النوعية من البرمجة جنباً إلى جنب مع برمجة الواجهات الرسومية فهي توفر آلية الربط بين الواجهات الرسومية والأكواد البرمجية وقد تكلمت عن أهم مبادئ هذه الطريقة في البرمجة في الفصل الثالث "أنت تعرف الكثير! اكتب برامجك الشخصية"، حيث سنستعرض كيفية إنشاء برنامج عملي يبنى بثقل على هذه الطريقة.

البرمجة الشيئية (الكائنية) Object Oriented

الطريقة المتداولة للبرمجة والأكثر شهرة في أوساط الشركات الكبيرة حتى الجامعات هذه الأيام تركز عليها غالباً. البرمجة الشيئية ليست مفهوماً جديداً بل لها من القدم والتاريخ ما يشهد على نموها وتبلورها على مدى عدة لغات برمجية تبنت هذه النظرة، ولكن تتبع الجانب التاريخي ليس مكانه هذه العجالة.

قبل أن نتكلم قليلاً عن البرمجة الشيئية يجب أن نعرف لماذا لاقت هذه الطريقة في البرمجة نجاحاً واسعاً بحيث أن اللغة التي لا تدعم هذه الرؤية البرمجية ولا تتيح الأدوات اللازمة لكتابة برامج شيئية تعتبر ناقصة في نظر الخبراء والشركات بشكل عام؟

ولهذا نرى أن حتى اللغات التي لم تكتب أصلاً لتدعم النموذج الشيئي، في وقت لاحق يضاف إليها دعم هذا النموذج، وكمثال لغة البرمجة بيرل التي هي لغة إجرائية في الأصل ولاحقاً أُضيف النموذج الشيئي للغة ولغة PHP وغيرها الكثير الكثير من اللغات التي أضافت دعم البرمجة الشيئية في فترات متأخرة من مسيرتها.

وليس ما يهمنا هنا هو كيف تطبق كل لغة مفهوم الكائن؟ وماذا يعني الكائن خلف الستار، ولكن السؤال المهم هنا، لماذا كل هذا الاهتمام بالبرمجة الشيئية مع أن كثير منا دخل عالم البرمجة من منطلقات مختلفة؟ قد يكون المنطلق الإجرائي أكثر قرابة للبداهة! لماذا هناك مباهاة ومقارنات بين مستوى دعم كل لغة للبرمجة الشيئية؟ وهناك سؤال مهم آخر هل الأفضلية للغات البرمجة الشيئية الصرفة مثل سمول توك Smalltalk وإيفل Eiffel أو للغات الهجينة الأخرى التي تتيح نوعاً من المرونة لا تجبر المبرمج على أن ينظر إلى كل شيء على أنه كائن؟

شخصياً أفضل الطريق الأخير لأنه حتى لغات البرمجة الكائنية الصرفة تختلف في تطبيق مبادئ ومفاهيم البرمجة الشيئية فلكل وجهة نظر وروى لمعالجة الموضوع محل الخلاف، لذلك المرونة دائماً سلاح جيد لكل لغة برمجة، ولا أقصد المرونة بمعناها الضيق مثلاً في الإعلان عن المتغيرات بل أريد المعنى الأوسع وهو حرية المبرمج في معالجة المشكلة بالطريقة والرؤية التي يختارها فتكون اللغة أداة مساعدة وليست عقبة.

لذلك أحب لغات البرمجة كلغة بيرل التي تجعل من الحرية شعاراً لها وتوفر للمبرمج عدة طرق لحل المشكلة يختار منها ما يشاء ويبدع ما يشاء بدون أن تجبره اللغة على طريق معين.

على أنه هنا أيضاً اختلاف والكثير يرون أن هذا الأمر سلاح ذو حدين وفي أغلب الأحيان يرجحون كفة الجانب السلبي فنسمع عبارات مثل «لا يمكن قراءة أكوادها» أو «تعطيك الحبل والكرسي لتشنق نفسك» وغيرها من العبارات التهكمية.

ويفضلون الشعار المغاير «هناك طريقة واحدة فقط لعمل هذا» وذلك باعتباره يفيد التنظيم ويساعد على بناء المشاريع العملاقة، لن أحول الموضوع هنا إلى مناقشة طويلة ولا انتصاراً لطرف

على حساب طرف آخر لكن سأستشهد بمقولة لمبرمج Lisp الكبير "بول غراهام" Paul Graham حيث يقول: «إننا كنا لا نعير الشركات التي تطلب مبرمجين جافا اهتماماً ولكننا كنا نخشى الشركات التي تطلب مبرمجين ليسب أو بيرل لأننا حينها نعلم أنهم يريدون مبرمجين حقيقيين!» وفي مكان آخر يقول: «إن سر نجاحنا هو استخدامنا Lisp ولكن يبدو أن لا احد مهتم بسرقة سر نجاحنا!». بعد كل هذا الاستطراد لنذكر شيئاً من مزايا البرمجة الشيئية:

١. **البساطة:** حيث الكائنات الوهمية تحاكي الكائنات الحقيقية، ذلك يؤدي إلى تقليل التعقيد والى هيكل برامج واضح للغاية سهل الفهم وسهل التمثيل على شكل رسومات توضيحية.
٢. **سهولة الصيانة:** حيث أن عملية الصيانة ستكون سهلة وسنعرف مكان الأخطاء بالتحديد لان كل كائن له استقلالية تامة.
٣. **إعادة الاستخدام:** حيث أن الكائنات يمكن إعادة استخدامها في عدة برامج وبذلك نوفر الوقت والجهد.
٤. **التطور:** حيث أن عملية التوسع ستكون سهلة من خلال إضافة أعداد قليلة من الكائنات الجديدة أو التعديل المباشر على الكائنات الموجودة، وذلك استجابة لأي تغييرات أو تحديات جديدة تفرضها بيئة التشغيل.
٥. **التكاملية وتقليل الاعتمادية:** حيث أن كل كائن يمثل كياناً منفصلاً، حيث أن العمل الداخلي للكائن منفصل عن أجزاء النظام الأخرى.
٦. **سهولة التعديل:** فهناك سهولة في إجراء أي تغييرات طفيفة في تمثيل البيانات أو الإجراءات في البرامج الشيئية. لان التغييرات داخل أي كلاس لا تؤثر على الأجزاء الأخرى للبرنامج، حيث أن الطريقة الوحيدة للعالم الخارجي في الاتصال بهذا الكلاس هو عن طريق واجهته باستخدام ميثودز الكلاس نفسه.

الآن ما هي البرمجة الشيئية؟

هي بكل بساطة عملية محاكاة الواقع في البرمجة فهي النظر إلى الوجود على أنه مجرد أشياء أو كائنات Objects ومن ثم محاكاته في عالم البرمجة. كل شيء في هذا الوجود يمكن أن ينظر إليه على أنه كائن وكل كائن له خصائص (سمات) Attributes وأفعال Methods. أنواع الخصائص Field type ثابتة كأن تكون قيمة رقمية أو نصية ولكن قيم الخصائص Field Value يمكن أن تختلف من حالة إلى أخرى. الكلاس Class هو عبارة عن قالب ننشئ منه الكائنات حيث داخل كل كلاس سيتم تعريف خصائص وأفعال كل كائن من هذا الكلاس مع الانتباه إلى أنه عادة لا يعتبر الكلاس كائناً في حد ذاته إلا في اللغات الكائنية الصرفة. إذاً عرفنا الآن أن الكلاس هو مجرد قالب جاهز لصنع كائنات من نوع ما.

نأتي الآن لنشير إلى وجود نوع آخر من الكلاسات وهو ما يسمي بالكلاسات المجردة Abstract Class هذه الكلاسات مهمتها تنظيمية فقط ولا يمكن إنشاء أي كائن منها مباشرة بل يجب أن يكون هناك كلاس يرث الكلاس المجرد وأن لا يكون هو بدوره كلاساً مجرداً حينها فقط يمكن أن ننشئ كائنات تحتوي على خصائص الكلاس المجرد. هنا ذكرنا مصطلح "الوراثة" Inheritance وهو بكل بساطة عملية انتقال كل خصائص وأفعال الكلاس الأب إلى الكلاس الابن. لنأخذ بعض الأمثلة قبل أن نواصل.

لنتخيل أن هناك مصنعاً للسيارات ونريد أن نحاكيه بالرؤية الشيئية يمكن أن نبسط إلى شيء مشابه إلى:

```
abstract class Vehicle
abstract class Sedan
class Camry
```

نلاحظ أننا أنشأنا سلسلة من الكلاسات لتنظيم الأمور حيث أنه بدأنا بكلاس مجرد باسم Vehicle (مركبة) وفيه سنعرف الخصائص والأفعال الأساسية التي يجب أن تحتويها كل مركبة ثم أنشأنا كلاس مجرد باسم Sedan يرث كل ما في كلاس Vehicle ويضيف عليها خصائص وأفعال كل عربة من نوع سيدان (صالون) أخيراً أنشأنا كلاس باسم كامري يرث كل خصائص سيدان ويضيف عليها مميزات وهوية الكامري التي نعرفها.

هنا يطرح السؤال لماذا نجعل من سيدان و Vehicle كلاسات مجردة؟ لماذا لا نتركها كلاسات طبيعية أليس الغرض هو الوراثة فقط؟ فيكون الجواب بكل بساطة لأننا نعلم مسبقا أننا لن ننشئ أي كائن من هذه الكلاسات وهذه عادة برمجية جيدة يجب الانتباه لها وعلى المستوى العملي يتخذ القرار فيها وقت التخطيط Design.

نلاحظ أن المخطط الآن قابل جداً للتطويع والتعديل في وقت لاحق مثلاً بإضافة مزيداً من أنواع السيارات بل وفئات السيارات. في المثال أعلاه كان المخطط ينحو منحى عمودياً بطبيعته أي سلسلة من الأعلى إلى الأسفل ولكن مع محاكاة أمثلة أكثر تعقيداً سيبدو شكل المخطط وكانه شجرة ذات غصون متفرعة تصلح لأن تحاكي الظواهر الطبيعية قبل الحقائق البرمجية كشجرة الكائنات الحقيقية من ثدييات...إلخ.

قبل أن أنتقل للنقطة القادمة وبما أن الكلام عن المخططات لا بأس بأن نذكر لغة النمذجة الموحدة UML والتي عادة ما تدرس جنباً إلى جنب مع مقرر لغات البرمجة الشيئية وهي لغة تمثيلية مفيدة جداً في مرحلة التخطيط وتحتوي على ١٤ نوعاً من المخططات تقع تحت مظلة تصنيفين رئيسيين:

١. مخططات هيكلية تركز على عناصر النظام وماذا يجب أن يحتوي.
٢. مخططات تفاعلية (تصف سلوك الكائنات) تركز على ماذا يجب أن يحدث في النظام المحاكى وكيف سيتم التواصل والتفاعل بين العناصر.

من خلال هذا الاستعراض البسيط بدأنا نعرف سر قوة البرمجة الشيئية في السيطرة على المشاريع العملاقة جداً، هذه القوة التي تنبع في حقيقة الأمر من التجريد.

في البرمجة الإجرائية عادة هناك مشكلة في كون البيانات مشاعة لكل أجزاء البرنامج ولكن في البرمجة الشيئية هناك نوع من الكبسلة (التغليف) Encapsulation أي أن البيانات الخاصة بأي كائن لا يمكن تعديلها إلا من خلال إرسال الرسائل Messages إلى الكائن وهو بدوره يقوم بعمل اللازم سواء بالتعديل المباشر أو إعادة إرسال رسالة أخرى إلى كائن آخر طلباً للمساعدة في إتمام الطلب.

من هنا نعرف أنه في البرمجة الشيئية تتم المهام عن طريق الرسائل ومجموع الرسائل التي يفهمها الكائن تسمى بروتوكولاً Protocol بعبارة أخرى البروتوكول أو الرسائل التي يمكن للكائن أن يتجاوب معها هي ذاتها الأفعال Methods المعرفة سلفاً في كلاس الكائن.

نعود هنا للإشارة إلى أهمية التجريد هنا أيضاً فانا كمستخدم لو أردت أن أرسل أزهاراً إلى صديقي ما علي سوى أن اذهب إلى محل الأزهار وأخبره بأنني أريد هذا النوع من الأزهار لكي يُرسله إلى صديقي.

هنا نلاحظ أنني أعطيت البائع رسالة تمثل طلباً ولا حاجة لي بمعرفة كيفية تعامل البائع لإتمام عملية الإرسال. هذه العملية تعرف بالصندقة السوداء Black Boxing أي أن كل كائن مسؤول عن التعامل مع كل رسالة أو طلب بطريقته الخاصة التي لا يجب أن يعرف عنها بقية الكائنات أي تفاصيل إضافية.

فبائع الزهور ربما يرسل طلب نقل الزهور إلى محطة البريد أو إلى شركة خاصة أو ربما يعطيها لصديق سيذهب إلى منطقة صديقي! وبما أننا نتكلم عن الرسائل لنتكلم عن التعددية Polymorphism وهي من أهم مميزات البرمجة الشيئية ونلخص بأنها قدرة استجابة أنواع مختلفة من الكائنات إلى رسالة واحدة ولكن كل كائن يفسرها بطريقته الخاصة. المثال المشهور الذي سيوضح لنا هذا المفهوم هو لو أن احد مدراء الشركات عقد اجتماعاً وبعد انتهاء الاجتماع قال لموظفيه تابعوا أعمالكم فالرسالة هنا واحدة ولكن كل موظف سيستجيب بطريقته الخاصة فموظف التسويق سيعود إلى محل البيع وموظف الدعم الفني سيعود إلى مكتب الدعم الفني وهلم جرا.

الآن بما أننا نفهم جيداً ميزة التعددية لنعد إلى الأفعال Methods حيث يمكن أن تعايش حالات أقوى من التعددية بدلاً من مجرد التشارك في أسماء الرسائل كما في المثال السابق. الحالة الأكثر انتشاراً هي ما يعرف بالأوفرلود Method Overload وهي قدرة الكائن على الاستجابة لرسالة معينة بعدة أشكال مختلفة على حسب نوع وعدد "الخيارات" Parameters المرفقة مع الرسالة.

مثلاً:

```
jump();
jump(Int myNumber);
```

فهنا لو أعطانا المرسل عدد مرات القفز في الرسالة لاستجاب الكائن بتكرار عملية القفز عدد المرات المطلوبة ولكن لو ترك المرسل رسالة فارغة من أي رقم ستكون الاستجابة للميثود الافتراضي ولنقل أنه يقوم بقفزة واحدة فقط. هذا مثال بسيط ولكنه يوضح فكرة جيدة أخرى وهي توفير أنواع مختلفة من الميثود لمعالجة حالات مختلفة من الرسائل والطلبات.

الحالة الأخرى للتعددية هي ما يعرف بالميثود أوفررايد Method Override وهي شكل من أشكال التعددية يحدث في حال لو كان هناك ميثود في الكلاس الأب ويرثه الكلاس الابن ولكننا نريد للكلاس الابن أن يستجيب بطريقة مغايرة لما يفعله الكلاس الأب. هنا نستخدم الأوفررايد حيث نقوم بإضافة ميثود في كلاس الابن يحمل نفس اسم وتوقيع Signature الميثود في كلاس الأب ولكن في جسم الميثود Method body نقوم بإضافة أكواد مغايرة لما هو موجود في كلاس الأب. مثال بسيط:

```
public class DaddyClass{
public void aMethod(){
//do something here
}}
public class myClass extends DaddyClass {
public void aMethod(){
//respond differently from my daddy
}}
```

تعدد التوارث Multiple Inheritance

في لغات البرمجة مثل ++C وبيرل هناك إمكانية لاي كلاس بأن يرث من أكثر من كلاس، هذا الشيء لا يخلو من الفائدة الكثيرة على الرغم من بعض العيوب ولكن في جافا مثلاً لا يمكن للكلاس أن يرث إلا من كلاس واحد وهنا يظهر أنه سنلاقي صعوبة ومحدودية ولكن جافا توفر ما يسمى بالواجهات Interface حيث يمكن لأي كلاس أن يطبق أكثر من واجهة بالإضافة إلى وراثته من كلاس معين. عندما يعلن كلاس بأنه يطبق واجهة ما فإننا ننشئ في الحقيقة عقد يجب من خلاله على الكلاس المطبق أن يوفر التطبيق البرمجي لكل ميثود موجود في الواجهة.

في البرمجة الشيئية كثيراً ما نتكلم محددات الوصول Access Modifier والتي من خلالها نضيف بعد آخر من إخفاء المعلومات Data Hiding طبعاً لكل لغة برمجية أنواع تحكم مختلفة ولكن في جافا مثلاً لدينا:

- private
- protected
- default
- public

طبعاً استخدام هذه الكلمات المفتاحية بالشكل الصحيح يحتاج خبرة وقرار المبرمج نفسه ولكن القاعدة العامة هي كلما كان هناك تشديد في الوصول إلى القيم والميثودز (الأفعال) كان أفضل فنحن قلنا أن كل كائن يجب أن يكون مسؤولا عن نفسه فلا يفضل التعديل على المعلومات من أماكن كثيرة ومختلفة تؤدي بدورها إلى المشاكل وصعوبة كتابة اختبارات للبرنامج.

البرمجة الوظيفية Functional Programming

البرمجة الوظيفية تعامل الحوسبة كتقييم للدوال الرياضية وتتجنب البيانات المشتركة Mutable والحالة State. تجب الإشارة هنا إلى أن البرمجة الوظيفية هي مجموعة أفكار إن صح التعبير وليست قوانين يجب متابعتها لذلك هناك اختلاف في تطبيق البرمجة الوظيفية فهناك اللغات الوظيفية الصرفة Pure Functional كلغة البرمجة هاسكل وغيرها حيث أنها لا تتيح إمكانية التغيير أو تحديث قيمة المتغير. بناءً على هذا الشرط تكون المتغيرات كالمتغيرات في الرياضيات حيث ص + ٢ = ٣ ولكن يمكننا أن نقول ض = ص + ٣ ولكننا سننشئ متغيرا آخراً لأن ص قيمة ثابتة غير قابلة للتغيير.

ولكن بشكل عام اللغات التي تدعم التالي يمكن أن تسمى وان تتيح البرمجة الوظيفية ولو إلى مستويات معينة.

- Code References
- Closures

تعمدت أن لا أتكلم عن Subroutines أو ما يعرف أيضاً باسم Functions والدوال في معرض الكلام عن البرمجة الإجرائية مع أن هذه الدوال تستخدم بكثرة في البرمجة الإجرائية كأداة لاستدعاء بعض الأكواد التي يتكرر استخدامها في البرنامج. مثال بسيط هو بدلاً من أن نضع أكواد تذييل صفحة ويب في كل صفحة يمكننا بكل بساطة حفظ وكتابة الأكواد داخل دالة مرة واحدة فقط والاكتفاء باستدعاء الدالة أينما احتجنا لها.

حسنا السؤال الآن إذاً كانت اللغات الإجرائية توفر الدوال فلماذا لا تسمى كلها لغات وظيفية؟ الجواب كي نستطيع أن نقول عن لغة أنها وظيفية يجب أن تتوافر فيها بعض المميزات التي ذكرنا بعضها في البداية وسنتطرق لبعضها الآخر بقليل من التفصيل فيما سيأتي في هذا الفصل.

المراجع References

أشرنا إلى هذه النقطة كنقطة أساسية للبرمجة الوظيفية وإن كانت ليست في حد ذاتها خاصية في البرمجة الوظيفية إلا إنها آلية تتيح لنا إمكانية البرمجة الوظيفية في بعض اللغات الإجرائية.

الآن ما هو المرجع Reference؟ هو من اسمه عبارة عن إشارة إلى مكان المعلومات. بعبارة أخرى يمكننا أن نقول أن المُشير أو المرجع هو متغير لكنه لا يحمل القيمة الحقيقية إنما يشير إلى مكانها فقط. كمثال عندما أقول تكلمنا سابقاً عن البرمجة الإجرائية في أحد الصفحات السابقة فأنا هنا أشير إلى مكان المعلومة في الكتاب ولكن ليس لدي المعلومة نفسها. في بيرل إضافة المعامل "\" قبل اسم المتغير ينشئ لنا مؤشراً، مثال:

```
my %hash = (one => 'hello', two => 'world');
my $hash_reference = \%hash;
```

نلاحظ هنا أن المتغير الذي يحمل القيم الحقيقية هو hash أما المتغير hash_reference فهو مجرد متغير يشير إلى مكان المعلومات ولا يحمل المعلومات بنفسه. والكلام هنا يطول ويتعقد كثيراً باختلاف لغات البرمجة لذا سنتجنب الإطالة ونكتفي بالفكرة العامة.

الآن لكي نتعرف أكثر على بعض المفاهيم المتداولة في البرمجة الوظيفية نحن بحاجة أولاً لان نعرف كيفية كتابة الدوال بشكل عام، الأمر الذي سيتيح لنا فهم المفاهيم الأكثر صعوبة.

إذن نبدأ بمثال بسيط عبارة عن طباعة رسالة تحية.

```
sub hello {
print "Welcome to my Website"}
```

في هذا المثال البسيط قمنا بإنشاء دالة باسم hello تقوم بطباعة رسالة ترحيبية في أي مكان تستدعى فيه، ويمكن استدعاؤها في أي مكان من البرنامج بهذا الشكل:

```
hello()
Parameters
```

قد يتبادر تساؤل عن مغزى تواجد القوسين الذين بعد اسم الدالة؟ هذه الأقواس تستخدم لتمرير قيم إلى داخل الدالة وبذلك نضفي ديناميكية إلى دالتنا فهي الآن تستطيع أن تنتج قيم مختلفة بحسب القيم الممررة. إذاً لنعدل المثال قليلاً:

```
sub hello {
$name = shift;
print "welcome to Ali's Website
";
print "welcome $name";}
hello("Ali");
```

جميل الآن نلاحظ أننا وضعنا قيمة بين القوسين لكي نستخدمها لاحقاً في داخل الدالة وهناك عدة طرق لكيفية استقبال القيم تختلف من لغة إلى أخرى ولكن في مثالنا Shift ستسند القيمة الممررة إلى المتغير Name وبذلك ستكون نتيجة الدالة طباعة جمل مختلفة بحسب اسم الزائر المدخل.

وإذا أحببنا إضافة المزيد من القيم الممررة ما علينا إلا أن نستقبلها كقائمة ونسندها لمتغيرات حسب الحاجة، مرة أخرى هذا الأمر يختلف من لغة إلى أخرى ولكن هذا نموذج:

```
hello("visitor"،"visitor2");
sub hello{
my ($name1, $name2) = @_;
print "welcome to Ali's Website
";
print "welcome $name1
";
print "welcome $name2";}
```

نلاحظ الآن أننا نستطيع أن نتعامل مع العديد من القيم الممررة بهذه الآلية. في بعض الأحيان لا نريد فقط أن ننفذ أوامر بل نريد من الدالة أن ترجع لنا قيمة لكي نسندها إلى متغير مثلاً أو ربما كي نستفيد من القيمة المرجعة في تقييم شرط من حيث الصحة.

كل ما علينا فعله الآن هو أن نستخدم كلمة Return وهذا المثال البسيط أدناه يشرح الفكرة:

```
sub sqr{
my $number = shift;
return $number * $number;}
my $squared = sqr(3);
```

ففي هذه الحالة عندما نمرر قيمة إلى دالة التربيع تقوم بإرجاع مربع الرقم فقط دون أن تطبعه إلى الشاشة كما كنا نعمل سابقاً، وكما نرى في هذا المثال أسندنا القيمة المرجعة إلى متغير لمزيد من العمليات لاحقاً في البرنامج.

المدى Scoping

من الأمور المهمة جداً في البرمجة ما يعرف بالمدى أو مدة حياة المتغير. في أغلب لغات البرمجة مدى المتغير محصور بين الأقواس المعقوفة مثلاً دائرة تكرار أو جملة شرطية أو حتى كائن أو Package...إلخ مما يعني أن المتغير خاص بالمنطقة المعرف بها ولا يمكن استخدامه خارجها، ولكن هناك متغيرات تكون عامة يمكن الوصول إليها من أماكن متعددة مما يجعلها عرضة للتغيير بشكل مستمر، ماذا لو كان اسم المتغير العام هو ذاته اسم المتغير الخاص؟ ستتولد لدينا مشاكل في المدى ونحصل على نتائج تكون غير متوقعة، لندرس هذا المثال:

```
$total = 10;
$number = 12;
adding($number);
sub adding{
$number = shift;
$total = $total + $number;
--$number;
}
print "$number، $total";
```

الآن ماذا تتوقعون ستكون النتيجة عندما نطبع قيمة المتغيرين؟

من المفترض أن يكون الناتج ٢٢ للمجموع لأننا جمعنا ١٠ و ١٢ هذا من جهة ومن جهة من المفترض أن تكون قيمة number كما هي. ولكن المفاجأة أن النتيجة ستكون صحيحة للمجموع ولكن قيمة Number ستكون ١١! لأننا داخل الدالة قمنا بإنقاص قيمة متغير اسمه Number أي أن المتغير الداخلي له نفس اسم المتغير الخارجي فأثر عليه، هذه النقطة تسبب الكثير من الأخطاء في اللغات التي تجعل من المتغيرات مشاعة Global افتراضياً.

إذن؛ يجب أولاً أن نجعل متغيرات كل دالة متغيرات خاصة بها قدر الإمكان بحيث لا يكون لها تأثير جانبي Side effects على متغيرات البرنامج في المقابل أيضاً يجب أن نحرص على أن لا نجعل من متغيرات البرنامج متغيرات عامة فهذا مصدر لكثير من المشاكل. في أكثر اللغات هذه ليست مشكلة فالمتغيرات ليست عامة ابتداءً ولكن في لغات مثل بيرل يجب أن نستخدم use strict للحصول على هذه الميزة ومن ثم نقوم بالإعلان عن كل متغير باستخدام my وبذلك نتفادى تعارض تأثير أسماء المتغيرات خارج مداها، لنعدل الكود إذاً:

```
my $total = 10;
my $number = 12;
adding($number);
sub adding{
my $number = shift;
$total = $total + $number;
--$number;
}
print "$number، $total";
```

الآن سنحصل على النتيجة المتوقعة ٢٢ و ١٢.

العودية Recursion

العودية هي عندما نستخدم ونطبق دالة في داخل تعريفها أو بعبارة أوضح عندما تستدعي الدالة نفسها. وفي البرمجة الشيئية عندما يستدعي الميثود نفسه يسمى Recursive Method. أقرب مثال لتوضيح الفكرة هو طريقة حساب Factorial لرقم معين والذي يتم عن طريق ضرب العدد في الأعداد الأصغر منه مثلاً للعدد ٥ تكون العملية: ٥*٤*٣*٢*١ = ١٢٥.

لبرمجة هذه العملية بالعودية نلاحظ النمط هو العدد الأصلي ضرب العدد الأصلي - ١ وهكذا إلى أن نصل إلى الواحد بناء عليه نوقف عملية الطرح ونبدأ العملية عكسية وعودية حيث أن الواحد قيمة معلومة ضرب الدالة التي كانت في الانتظار وهكذا، إذاً ما سيحدث داخل الدالة هو بالترتيب:

```
5*f(5-1)
4*(4-1)
3*(3-1)
2*(2-1)
f(1) = 1
```

عندما نصل إلى هذه النقطة سترجع الدالة قيمة معلومة وليس استدعاء للدالة مرة أخرى عليه بما أن القيمة معلومة نضربها مع الاثنين...إلخ.

وهذه الأكواد البرمجية للدالة:

```
$fv = fact(5);
print "factorial 5 is $fv
";
sub fact {
my $var = shift;
if ($var > 1){
$fv = $var * fact($var -1);}
else {$fv = 1;}
}
```

هذه الظاهرة لها استخدامات كثيرة ومفيدة وكمثال أخير دالة لتحويل الأرقام إلى ما يمثلها من Binary وذلك باستخدام العودية.

```
sub binary {
my ($n) = @_;
return $n if $n == 0 || $n == 1;
my $k = int ($n/2);
my $b = $n % 2;
my $E = binary($k);
return $E.$b;
}
print binary(100);
```

وكما شاهدنا في المثالين وفي أي حالة أخرى يجب توفير نقطة معلومة لإيقاف عملية الاستدعاء المتكرر وإلا لن تتوقف عملية الاستدعاء كما هي الحال في دوائر التكرار اللانهائية.

الدوال العالية Higher-order functions

الآن سنبدأ استعراض بعض الخصائص الحقيقية للبرمجة الوظيفية، وسنبدأ بالدوال العالية حسب اصطلاحهم وربما يعرفها البعض باسم Callbacks أو Factories. وتسمى الدالة بهذا الاسم عندما تحقق أحد أمرين:

١. أن تستقبل دالة كمدخل (قيمة ممررة).

٢. أن ترجع دالة بدلاً من قيمة معينة.

وقبل أن نأخذ بعض الأمثلة يجب أن نتعرف على بعض النقاط.

الوظائف الفرعية المجهولة Anonymous subroutines

وهي دوال مجهولة أو بعبارة أخرى ليس لها اسم (معرف Identifier)، مثلاً:

```
my $number_plus_two = sub {return shift()+2};
print $number_plus_two->(3);
```

في هذا المثال نلاحظ أننا أنشأنا متغير ولكن لم نسند له قيمة بل أسندنا له دالة والمهم هنا نلاحظ أنها لا تحتوي على اسم. الآن نلاحظ عند تمرير قيمة ٣ سيتم جمعها مع ٢ لتكون النتيجة ٥. إذاً نستطيع أن نقول للدوال المجهولة فوائد منها:

١. أننا نستطيع إسنادها إلى متغير أو قائمة أو هاش.

٢. نستطيع بعدها أن نمررهم إلى دوال أخرى على شكل Arguments.

٣. سنحتفظ بالمتغيرات في المدى المحيط.

٤. نستطيع إنشاءهم في وقت التنفيذ.

المغلقات Closures

لندرس هذا المثال:

```
sub demo{
my $name = shift;
return sub{print "Hello $name"}
}
my $ref = demo("Ali");
&$ref();
```

الأمر الملفت النظر هنا هو أننا نلاحظ أن الدالة المجهولة استطاعت أن تصل إلى متغير غير معرف في مداها بل موجود في الدالة المحيطة بها. هذه الظاهرة تسمى Closure. مثال أكثر تقدماً:

```
sub demo{
my ($title) = @_;
return sub{my ($name) = @_;
print "$title $name"}
}
my $ref = demo("Mr.");
my $ref_2 = demo("Ms.");
```

الآن نلاحظ النتيجة:

```
$ref->("Ali"); #Mr. Ali
$ref_2->("Yosra"); #Ms. Yosra
```

نلاحظ كيف أن الدالة احتفظت بالمدخلات في المرحلة الأولى ومن ثم طبعت المتغيرين بالشكل الصحيح.

سنحاول أن نواصل الكلام في هذا الباب في الإصدارات القادمة إن شاء الله وسنكمل الكلام بتفصيل أكثر وسنتناول مواضيعاً مثل: Currying, Lazy eval, Streams, Continuation ... وغيرهم.

البرمجة التعريفية (إعلانية) Declarative

هي وصف لمنطق الحوسبة بدون وصف طريقة التحكم بمجراها وبذلك تكون عكس نوع البرمجة Imperative حيث كنا نخبر البرنامج بطريقة حل المشكلة خطوة بخطوة. أي أن الاهتمام هنا هو بالنتيجة وليس الطريقة، وسنترك الكلام في بعض أساسيات هذا النموذج في إصدار آخر من هذا الكتاب أن شاء الله.

مجموعة من المصطلحات المتنوعة

IDE

هذه الكلمة تمر علينا كثيراً وهي اختصار Integrated Development Environment، أي بيئة التطوير المتكاملة، ولكن ماذا يعني هذا بالضبط؟

في الماضي وإلى الآن كان بالإمكان الاكتفاء بمحررات النصوص مثل المفكرة في ويندوز و vi في لينكس للبرمجة، حيث يمكن لنا أن نبرمج برامج كاملة فقط باستخدام المفكرة البسيطة ثم نقوم بعملية الترجمة للأكواد، هذه الطريقة وان كانت جيدة من ناحية أنها تجبر المبرمج على الاعتماد على نفسه في البرمجة بدون تدخل من البرنامج للمساعدة، إلا أنها تقلل الإنتاجية في أحيان كثيرة خاصة عندما يكون هناك حاجة للتعامل مع أدوات أخرى مثل متقصي الأخطاء ومدير المشاركة ومدير الاختبارات...إلخ.

طبعاً لا يزال هناك من يعتقد أن برامج كتابة الأكواد يجب أن تكون بسيطة في حد ذاتها ولا تغرق المبرمج في بحار الخصائص التي في الغالب لن يستخدمها بذلك الشكل المستمر. حتى عهد قريب كنت أتبنى هذا الرأي لأنني كنت أرى أن المبرمج فعلاً يجب أن يكون خبيراً في لغته وحافظاً لكل تفاصيلها وخفاياها فلا يحتاج إلى مساعدة من أي أحد حتى لو كان برنامج التطوير وكنت أتخذ من هذه الطريقة وسيلة لكي أصقل مهارتي، حيث كنت في ويندوز أتعامل مباشرة مع المفكرة، ثم تطور الأمر إلى استخدامي لبرنامج Padre والذي كان بسيطاً جداً في إمكانياته وقتها ولكن أحببته لأنه يقدم خدمة تلوين الأكواد. كنت أتجنب كل البرامج التي تنطوي تحت مسمى IDE مثل Eclipse و Netbeans بل حتى Emacs لأنني كنت أومن بأن الفترة التي سأقضيها في تعلم هذه البرامج واكتشاف خفاياها وحفظ اختصاراتها، يجب أن استغلها في تعلم لغتي المفضلة.

على كل حال في يوم ما بدأت العمل على مشروع يعتمد على لغة البرمجة جافا وكان يتوجب علي عمله باستخدام Eclipse ومن خلال عملي في هذا المشروع رأيت أن الفترة التي احتجتها لتعلم البرنامج لم تكن بتلك الطول هذا فضلاً عن الفوائد التي تحصلت عليها من استخدام Eclipse، خاصة فيما يتعلق بلغة البرمجة Java، عندها بدأت أومن ببيئات التطوير المتكاملة مع بعض التحفظات على بعض النقاط التي إلى الآن أفضل أن أعملها بنفسي مثل إنشاء الواجهات الرسومية فهذه البرامج تصلح لتصميم الشكل الأولي بشكل سريع ولكن عندما أريد أن اكتب الواجهة النهائية في الغالب سأكتبها بنفسي، وكذا الحال أيضاً في كتابة أكواد HTML فالتعامل مع الأدوات المرئية في المجال تطوير لا أرى له ضرورة في أغلب الأحيان.

على كل حال هذه مجرد تجربة شخصية، نعود إلى بيئات التطوير المتكاملة ماذا تعني متكاملة هنا؟ نستطيع أن نقول أن التكامل هنا يعني أن نتحصل على برنامج واحد يقدم محرر نصوص وخدمات تحرير النصوص البرمجية مثل التلوين للأكواد والإكمال التلقائي وإبداء الاقتراحات وحتى القدرة على تصحيح الأخطاء الكتابية في الوقت الحقيقي مثلما يوجد في Eclipse. ثم يجب أن يقدم البرنامج مفسراً أو مترجماً للغة المعنية وقد يستخدم مفسراً خارجياً في بعض الحالات. أيضاً قد تحتوي البيئة على أدوات البناء التلقائي. أيضاً قد يحتوي البرنامج أداوت لعملية الاختبار والتجريب. وما إلى ذلك من خدمات أخرى تختلف من برنامج إلى آخر حسب إمكانياته مثل توفير أدوات رسم الواجهات الرسومية وأدوات العمل الجماعي...إلخ.

مُترجمة ومُفسرة Compiled and Interpreted

يمكن بشكل عام تقسيم لغات البرمجة إلى مفسرة أو مترجمة أو حتى مفسرة ومترجمة في آن واحد كما هي الحال في بعض اللغات. المترجم يحول الكود المصدري (ما نكتبه بلغة البرمجة) إلى كود لغة أخرى غالباً إلى لغة يفهمها الحاسب. عادة الهدف الأساسي من الترجمة هو الحصول على ملف تنفيذي. أما اللغات المفسرة فيتم إرسال الأكواد إلى مفسر اللغة والذي يقوم هو بدوره بتنفيذ الأكواد مباشرة. بين هذا وذاك توجد لغات لديها مفسر ومترجم وتوجد لغات مثل جافا يتم في البداية ترجمة الملف المصدري إلى لغة وسطية Bytecode ومن ثم يتم تفسير هذا الكود من خلال المفسر الذي هو بالنسبة للجافا هو ذاته الآلة التخيلية Virtual Machine. طبعاً توجد آلات تخيلية عديدة غير الخاصة بلغة جافا لديها القدرة على التعامل مع لغات متعددة مثلاً Parrot وهي الآلة التخيلية الجديدة التي تستهدف دعم اللغات الديناميكية مثل بيرل وبايثون وروبي...إلخ.

اللغات المفسرة لا يتم التدقيق على الأخطاء وقت الترجمة (البناء) كما حال اللغات المترجمة فعليه يجب أن يتم التدقيق على الأخطاء وقت التنفيذ وهذا يعني أن السرعة ستكون أقل وهناك تفاوت كبير بين أداء المفسرات من حيث السرعة. لكن الاعتماد على المفسر يعطي ميزة انتقالية Portability أكثر من المترجم للتنقل بين نظم التشغيل والأنظمة المختلفة.

تركيبية ودلالية Syntax and Semantics

في البرمجة دائماً ما يمر علينا هذان المصطلحان، فالأخطاء عادة ما تصنف إلى أنها أخطاء Syntax أي أخطاء إملائية (كتابية أو تتعلق بتراكيب الجملة بلغة البرمجة) لا تتبع القواعد العامة لكتابة الأكواد في لغة البرمجة المعنية، وهذه الأخطاء قد تكون بسبب أخطاء في كتابة الكلمات المفتاحية للغة، أو أخطاء في ترتيب الكتابة كالخطأ في مكان كتابة الشرط أو أخطاء نسيان إضافة قوس أو قوس معقوف وما إلى ذلك. إذاً فإن Syntax بالنسبة لأي لغة هو بعبارة أخرى قواعد كتابة الأكواد في اللغة وهنا يظهر تباين كبير بين لغات البرمجة فمن اللغات ما يهتم بأن يكون سهل القراءة ويستخدم طريقة واضحة خالية من الرموز والأقواس الكثيرة.

وهناك لغات يكثر فيها ظهور الأقواس المعقوفة والأقواس والرموز وغيرها، على أنه في الوهلة الأولى قد يتصور أن الطريقة الأولى هي الطريقة الأفضل فمن منا لا يحب أن تكون اللغة سهلة القراءة إلا أن الموضوع ربما يكون أعقد مما يبدو. فمثلاً الإسراف في توضيح أسماء المتغيرات والدوال في جافا أثار نقداً بسبب طول هذه الأسماء والتعب والتطويل في إعادة كتابتها مرات ومرات من جهة وصعوبة حفظها من جهة أخرى حيث أننا نجد حاجة ماسة لمساعدة بيئة تطوير متكاملة IDE لإظهار الاقتراحات وإكمال الأسماء.

في الجانب المقابل هناك لغات مثل بيرل الكثير ممن يقرؤون بعض أكوادها يصابون بحالة خوف من كثرة الرموز المتواجدة ولكن في حقيقة الأمر هذه الرموز لم تُضف عبثاً فعلى سبيل المثال رموز مثل $, @ وغيرها لم توضع إلا للاختصار ومساعدة المبرمج في أن يكتب أكواده بشكل مختصر وسريع. في Lisp هناك كلام كثير عن كثرة الأقواس فيها بين مؤيد ومعارض لذلك أرى أن هذا الجانب الذي قد يكون ثانوياً إلا إنه مؤشر جيد لاختيار لغة البرمجة التي ستبدأ بها، لأنه لا يخفى أن كثيراً من المبرمجين إنما يحبون ويتعصبون لبعض اللغات لأنهم يستمتعون بكتابة أكوادها.

نعود إلى Syntax ونقول أن هذه النوعية من الأخطاء أن ظهرت في البرنامج فهي ليست ذات أهمية قصوى لأنها عادة ما ستُصطاد في مرحلة بناء البرنامج وسيشير المترجم أو المفسر إلى مكان الخطأ تحديداً، عندها يمكننا أن نرجع إلى مكان الخطأ ونصححه بسهولة تامة. هذا فضلاً عن أنه هذه الأيام توجد IDE's مثل إكلبس وغيره لديها القدرة على اصطياد الأخطاء الطباعية في الوقت الحقيقي مباشرة.

أما Semantics فهي الأخطاء المعنوية (المنطقية) في البرنامج، مثلاً لماذا برنامج يرجع قيمة مغايرة عن القيمة المتوقعة؟ هذه الأخطاء لن يشير إليها المترجم أو المفسر وسيعمل البرنامج بشكل طبيعي لذلك عادة ما تكتشف مثل هذه الأخطاء في وقت التشغيل والتجريب، أيضاً هذه النوعية من الأخطاء تعرف بالحشرات Bugs، وقد يطول أمر اكتشافها إلى شهور إذا كانت معقدة وخفية.

قبل أن نختم هذه الفقرة يجب أن ننوه بمصطلح Syntactic Sugar وهو بكل بساطة إضافة المزيد من التراكيب Syntax إلى لغة برمجة معينة بحيث تزيد من سهولة كتابة الأكواد وتيسر على المبرمجين قراءة الأكواد، ولكن مع كل ذلك فإزالتها مرة أخرى من اللغة يجب أن لا تحدث أي أثر على قدرة اللغة الأساسية. بعبارة أخرى هي كلمات وقواعد تضاف إلى اللغة كي تساعد على التعبير عن بعض الخصائص بشكل أكثر تركيزاً أو أكثر وضوحاً أو بشكل مختلف ولكنها في حد ذاتها لا تضيف ميزة جديدة للغة البرمجة.

ولهذا نقرأ في ويكيبيديا: «لغات البرمجة العالية المستوى هي لغات آلة مع الكثير والكثير من Syntactic Sugar»، فعلاً هذه المقولة شديدة التعميم ولكن في منتهى الصحة. فنحن يمكننا أن نبرمج بلغة الآلة كل ما نستطيع أن نبرمجه باللغات عالية المستوى ولكننا نفضل الأخيرة لأنها تقدم أسلوب كتابة أسهل بكثير وأكثر إنتاجية للمبرمج. من الأمثلة لغة البرمجة C ليست لغة برمجة كائنية التوجه إلا إنه يمكن كتابة برامج شيئية باستخدام مؤشرات الدوال، و Type casting و Structures. ولكن لغة البرمجة ++C تسهل عملية البرمجة الشيئية وتجعلها أكثر أناقة بإضافة الأدوات المساعدة تراكيب الجُمل المناسبة للبرمجة الشيئية.

أيضاً مثال آخر أكثر وضوحاً هو إضافة Moose للغة بيرل حيث تحتوي على نظام كائني جيد ولكن طريقة كتابته تختلف كثيراً عما نراه في جافا وغيرها من اللغات الكائنية لذلك أُنشِأت Moose وهي أفضل طريق لتسهيل البرمجة الكائنية في بيرل للقادمين من لغات أخرى ومزاياها تفوق الحصر ولكن سأضيف هذا المثال لكي نرى كيف تكتب البرمجة الشيئية في بيرل بعد إضافة Moose.

إليك المثال التالي:

```
package User;
use DateTime;
use Moose;
extends 'Person';
has 'password' => (
is => 'rw',
isa => 'Str',
);
has 'last_login' => (
is => 'rw',
isa => 'DateTime',
handles => { 'date_of_last_login' => 'date' },
);
sub login {
my $self = shift;
my $pw = shift;
return 0 if $pw ne $self->password;
$self->last_login(DateTime->now());
return 1;
}
```

جمع المخلفات Garbage Collection

جمع القمامة في علوم الحاسب الآلي هو نوع من الإدارة الأوتوماتيكية (التلقائية) للذاكرة. أي أنها نوع خاص من إدارة الموارد، أي الذاكرة في مثالنا. وهي من اختراع جون مكارثي في سنة ١٩٥٩ ليحل مشاكل Lisp. حيث يقوم جامع القمامة باستعادة الذاكرة المحجوزة من قبل كائنات أو غيرها لم تعد مستخدمة من قِبَل البرنامج.

إذاً فجامع القمامة مهمته أن يجعل من عملية إدارة الذاكرة اليدوية أمراً تلقائياً بحيث لا يحتاج المبرمج بنفسه بأن يحرر الذاكرة المشغولة.

فلسفة عملها تتلخص بشكل مبسط في:

١. ابحث عن أي كائن لم يعد يمكن الوصول إليه في زمن تشغيل البرنامج المستقبل.
٢. استعد الذاكرة المحجوزة من هذا الكائن.

بعض اللغات توفر جامع قمامة والبعض الآخر لا توفر ذلك وتتركه للمبرمج وسنأتي على أمثلة لكلا النوعين لاحقاً.

الاستمرارية Persistence

هي الثبات والاستمرارية، وهي مجموعة المعلومات التي تستمر حتى بعد انتهاء دورة تشغيل البرنامج. مثلاً عندما يقوم المستخدم باللعب ويصل إلى مستويات متقدمة نحتاج إلى وسيلة كي نحفظ تقدمه في اللعبة وإلا سيضطر إلى أن يبدأ من البداية كلما أعاد تشغيل اللعبة. بدون هذه الخاصية ستعيش المعلومات فقط في الذاكرة العشوائية المؤقتة مما يعني ضياعها مع إيقاف تشغيل الكمبيوتر. ويمكن أن نصل إلى الاستمرارية والثبات من خلال حفظ هذه المعلومات على موارد الذاكرة الدائمة مثل الأقراص الصلبة أو قواعد البيانات الموجودة على خوادم مخصصة لهذا الغرض...إلخ، ولغات البرمجة توفر طرق كثيرة لحفظ البيانات مثل الكتابة إلى الملفات النصية وملفات XML وغيرهم.

الفصل الثاني

لمحة عن لغات البرمجة

١. Perl

لغة بيرل وهي اختصار لجملة Practical Extraction and Report Language ظهرت على يد المبرمج واللغوي المشهور لاري وال Larry Wall في سنة ١٩٨٦ حيث بدأت مرحلة التطوير بالتتابع إلى أن وصلت اللغة إلى مرحلة نضج عالية في الإصدارة الخامسة. ومنذ سنة ٢٠٠٠ بدأ العمل على بيرل ٦ والتي تعتبر إعادة كتابة كاملة للغة بالإضافة لمشروع Parrot (الببغاء) وهي المنصة التي ستعمل عليها بيرل مع القدرة أيضاً لإضافة عدة لغات أخرى.

تتمتع لغة البرمجة بيرل بأرشيف ضخم من الإضافات الجاهزة أو ما يسمى سيبان CPAN وهي عبارة عن مكتبات وسكربتات جاهزة للاستخدام وإضافة المزيد من القدرات إلى لغة البرمجة بيرل حيث يمكن إضافة Modules خاصة بمكتبات برمجة الألعاب وتصميم المواقع وغيرها من التطبيقات المفيدة.

اكتسبت بيرل شهرة واسعة بسبب قوتها في التعامل مع التعابير النمطية Regex، وكفائتها في إدارة المواقع العملاقة مثل Amazon و Slashdot وقاعدة بيانات الأفلام IMDb وغيرها. وللغة شعار مشهور "يوجد أكثر من طريقة لعمل هذا" وهذا يرجع إلى انسيابية اللغة والقدرة على التطويع مما أدى إلى التفنن في كتابة الأكواد وظهور شعار "Just another Perl hacker"، ونتيجةً لذلك ظهر بعض المتهكمين من مبرمجي اللغات الأخرى واتهموا الكتابة بهذه اللغة بالقبح.

لغات أثرت على لغة بيرل:

- Lisp
- Awk
- Sed
- C
- C++
- Smalltalk
- Pascal

لغات تأثرت بلغة بيرل:

- Python
- PHP
- Ruby
- Dao
- Javascript
- Falcon
- Windows PowerShell

خصائص اللغة:

١. مُفسرة.

٢. ديناميكية.

٣. عالية المستوى High level.

٤. إجرائية.

٥. وظيفية.

٦. تدعم البرمجة الكائنية.

٧. لا تعتمد على نظام تشغيل معين (Cross Platform).

أهم التطبيقات:

١. برامج الويب.

٢. إدارة الأنظمة والخوادم.

٣. إدارة قواعد البيانات والشبكات.

٤. معالجة الملفات النصية.

٥. برمجة الجرافكس.

الموقع الرسمي:

www.perl.org

ترخيص اللغة:

GNU General Public License, Artistic License

مثال برمجي:

برنامج تخمين رقم بين ١ و ١٠.

```
my $number = 1 + int rand 10;
do { print "Guess a number between 1 and 10: " } until <> == $number;
print "You got it!
";
```

٢. Java

لغة جافا Java هي في حقيقة الأمر عبارة عن امتداد للغة Oak، حيث ظهرت هذه اللغة في بداية التسعينات من قِبَل شركة صن ميكروسستمز Sun Microsystems لتكون لغة سهلة الاستعمال والتنقل وكانت تستهدف التواصل بين أجهزة التسلية مثل أجهزة الألعاب و VCR.

الهدف الأساسي كان استثمار اللغة في أجهزة التلفاز التي تقدم خدمة الفيديو حسب الطلب. وفي هذه الفترة بدأت الإنترنت بالرواج وقد لاحظ مطوري Oak هذه الموجة ومستقبلها خاصة بعد ظهور أول متصفح رسومي، فتحول اهتمامهم إلى الويب فأنشؤوا ويب رنر WebRunner (Hot Java Web Browser) وهو عبارة عن متصفح يدعم لغة برمجة اوك. في هذا الحين وجدوا أن اسم "Owk" مُسجل من قبل لشركة تدعى "Oak Technology"، لذا قاموا بتغيير اسم اوك إلى جافا. وتطور الأمر لاحقاً بمشاركة العديد من الشركات والمطورين لتكتسب جافا شعبية وقوة كبيرة.

لغات أثرت على لغة جافا:

- C++
- Ada 83
- Smalltalk
- Modula 3
- Oberon
- Objective C

لغات تأثرت بلغة جافا:

- Clojure
- PHP
- Python
- D
- Groovy
- Scala
- C#

خصائص اللغة:

- لا تعتمد على نظام معين.

حيث يقوم الكومبايلر الخاص بجافا بإنتاج بايت كود تحوله JVM إلى لغة الآلـة، فبرامـج جافـا تعمـل علـى أي نظام توجد فيه JVM.

- لغة برمجة كائنية.
- أسلوب الكتابة مشابه للغة ++C.
- تحتوي على جامع قمامة Garbage Collection.
- مكتبة قياسية غنية.

حيـــث تحـــوي عـــدداً ضـــخماً مـــن الكلاسات والميثودز ويمكن تصنيفها إلى ستة أقسام:

١. كلاسات دعم اللغة.
٢. كلاسات الأدوات (المساعدة).
٣. كلاسات الإدخال والإخراج.
٤. كلاسات الشبكة.
٥. AWT لبرامج الواجهة الرسومية.
٦. Applet لبرامج المتصفحات.

أهم التطبيقات:

- تطبيقات الويب.
- برامج سطح المكتب.
- برامج الشبكات.
- برامج الجوالات.
- تطوير الألعاب.

الموقع الرسمي:

http://www.oracle.com/technetwork/java/index.html

ترخيص اللغة:

GNU General Public License

مثال برمجي:

برنامج تخمين رقم بين ١ و ١٠.

```
public class Guessing {
public static void main(String[] args) throws
NumberFormatException{
int n = (int)(Math.random() * 10 + 1);
System.out.print("Guess the number between 1 and 10: ");
while(Integer.parseInt(System.console().readLine())! n){
System.out.print("Wrong! Guess again: ");
}
System.out.println("Well guessed! );
}
}
```

٣. C

قام دينيس ريتشي Dennis Ritchie من معامل بل Bell Labs في سنة ١٩٧٢ بتطوير لغة البرمجة المشهورة جداً C. سي تُبني وتقتبس كثيراً من سابقتها لغة B وسابقات لغة بي مثل CPl و BCPL.

لغة CPL طُورت لهدف أساسي هو أن تكون لغة برمجة عالية المستوى وألا تكون مرتبطة بمعالج أو نظام معين بالإضافة لإتاحة الفرصة للمبرمج للتحكم بالأمور التحتية Low Level. ولكن نقطة ضعف هذه اللغة كانت أنها كبيرة جداً في الاستخدام لعدة تطبيقات. وفي سنة ١٩٧٦ طُورت BCPL وهي عبارة عن نسخة مصغرة من CPL مع المحافظة على خصائص وأهداف اللغة الأم. وفي سنة ١٩٧٠ قام كين تومسن Ken Thompson من معامل بيل بإنشاء لغة B والتي هي الأخرى تصغير للغة BCPL مع هدف أساسي وهو برمجة النُظم. وفي نهاية المطاف قام دينيس ريتشي بإعادة بعض المزايا العامة من BCPL إلى B لتظهر إلى الوجود اللغة فائقة الشهرة C.

وعندما ظهرت قوة سي وقابليتها العالية للتطويع أُعيد كتابة نظام التشغيل يونكس Unix بشكل شبه كامل باستخدام سي وقد كان مبرمجاً في الأساس بلغة أسمبلي، وعلى مدى السبعينات انتشرت هذه اللغة في الجامعات والكليات لارتباطها بنظام يونكس وتوفر أدوات التصنيف Compilers الخاصة بها. ومع انتشار سي وتبني كل مؤسسة تطوير نسخة خاصة ظهرت مشكلة عدم التوافقية، مما حدى المعهد الأمريكي الوطني للمواصفات ANSI إلى تشكيل لجنة خاصة لتبني تعريف ومواصفات موحدة للغة سي.

لغات أثرت على لغة سي:

- ALGOL68
- FORTRAN
- B, BCPL, CPL
- Assembly
- PL/I

لغات تأثرت بلغة سي:

- C++
- C#
- JAVA
- Perl
- AWK
- Limbo

خصائص اللغة:

- لغة مُترجمة (تستخدم Compiler) تتميز بالسرعة الكبيرة.
- لغة تتيح الوصول إلى الأوامر والمهام التحتية Low Level مع أسلوب كتابة عالٍ المستوى فهي لذلك تصلح لبرمجة نظم التشغيل جنباً إلى جنب مع برمجة التطبيقات المعتادة.
- أغلب الميزات مفصولة من قلب اللغة وتضاف كمكتبات جاهزة للاستخدام.
- لغة مُنظمة وإجرائية.
- أسلوب الكتابة ثابت وضعيف.

أهم مجالات التطبيق:

- أنظمة التشغيل وبرامج يونكس.
- برمجة الألعاب.

ترخيص اللغة:

مترجم جنو الخاص باللغة.

GNU General Public License

مثال برمجي:

برنامج تخمين رقم بين ١ و ١٠.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
int n;
int g;
char c;

srand(time(NULL));
n = 1 + (rand() % 10);

puts("I'm thinking of a number between 1 and 10.");
puts("Try to guess it:");

while (1) {
if (scanf("%d", &g)! 1) {
/* ignore one char, in case user gave a non-number */
scanf("%c", &c);
continue;
}

if (g == n) {
puts("Correct! );
return 0;
}
puts("That's not my number. Try another guess:");
}
}
```

٤. Smalltalk

قامت مجموعة البحث التعليمي في زيروكس PARC بقيادة الآن كي Allan Kay بتصميم لغة برمجة كائنية عُرفت باسم سمول توك ٧٢. وبعد المزيد من التجارب والتطوير توالت نسخ أخرى إلى أن انتهت اللغة إلى سمول توك ٨٠.

لغات أثرت على لغة سمول توك:

- Lisp
- Simula
- Logo
- Sketchpad

لغات تأثرت بلغة سمول توك:

- Perl
- Ruby
- Python
- Java
- Objective C
- Self
- Falcon

خصائص اللغة:

- لغة برمجة كائنية صرفة.
- أسلوب الكتابة ديناميكي.
- لغة متعددة التطبيقات.
- تتوافر لها بيئة تطوير مرئية.

أهم مجالات التطبيق:

- نظم المعلومات الإدارية.
- مناسبة جداً للمشاريع العملاقة.
- برامج الباتش لبرامج MainFrame الكبيرة.
- تستخدم كلغة أكاديمية للتدريس في الجامعات.
- لديها القدرة على إدارة نظام الهاتف لدولة كاملة.

ترخيص اللغة:

مترجم جنو الخاص باللغة.

GNU General Public License v2.0

مثال برمجي:

```
'Hello World!' displayNl
```

٥. Lisp

مع منتصف خمسينات القرن الماضي ظهرت موجة الاهتمام بالذكاء الاصطناعي. نشأ هذا الاهتمام الكبير بسبب رغبة اللغويين بالتعرف على معالجة اللغات الطبيعية، وعلماء النفس بسبب محاولة محاكاة المعلومات الإنسانية، وأخيراً علماء الرياضيات بسبب الرغبة في جعل عملية إثبات النظريات عملية أوتوماتيكية. والشيء المشترك بين كل هذه التطبيقات هو الحاجة إلى طريقة تسمح للكمبيوترات بمعالجة البيانات الرمزية على شكل قوائم.

كانت شركة IBM من أول الشركات المهتمة بهذا المجال في أواسط الخمسينات. في نفس الوقت كان العمل جارياً على مشروع الفورترن. ولغلاء تكلفة إصدار أول مُصنّف للفورتران أُضيفت معالجة القوائم كإضافة مستقلة إلى الفورتران عرفت باسم FLPL.

في سنة ١٩٥٨ عمل جون ماكارثي John McCarthy في شركة IBM في إدارة بحوث المعلومات. تم استقدام مكارثي ليعمل على إنشاء مجموعة من متطلبات عمل الحوسبة الرمزية.

المحاولة الأولى كانت التفرقة بين التعابير الجبرية. هذه التجربة الأولية انتجت قائمة من متطلبات اللغة من أهمها التعابير الشرطية، والعودية. هذه المتطلبات لم تكن موجودة في أي لغة برمجة في ذلك الزمان حتى أعلى اللغات مستوى حينذاك وهي فورتران Fortran.

تعود بدايات العمل على ليسب إلى سنة ١٩٥٦ حيث قام جون مكارثي بتطوير أُسس لغة ليسب Lisp في مشروع صيف دارتموث للأبحاث. كان طموح مكارثي أن ينشئ لغة معالجة قوائم للذكاء الاصطناعي. وفي سنة ١٩٦٥ ظهرت أول إصدارة من ليسب.

ومع سنة ١٩٧٠ ظهرت حواسيب خاصة فقط بتشغيل برامج ليسب عرفت باسم "أجهزة ليسب". وفي مطلع الثمانينات أُدخلت مبادئ البرمجة الكائنية إلى ليسب. ومع سنة ١٩٨٦ بدأ العمل على توحيد معايير ليسب وفي سنة ١٩٩٢ نُشرت معايير ANSI Common Lisp.

في الوقت الراهن توجد إصدارات عديدة من ليسب من أهمها Arc, Common Lisp, Scheme بالإضافة إلى لغات مخصصة لتطبيقات معينة مثل Emacs Lisp. تعتبر ليسب في نظر الكثيرين أقوى وأجمل لغة برمجة على الإطلاق ويوجد الكثير من المتعصبين لها فمن أمثلة تلك المقولات «الله يبرمج باستخدام ليسب!» . وتعتبر البرمجة بها متعة للحرية التي توفرها للمبرمج كما تفعل بيرل.

لغات أثرت على لغة ليسب:

- IPL

لغات تأثرت بلغة ليسب:

- Perl
- Ruby
- Python
- Javascript
- Forth
- Mathematica
- Falcon
- Lua
- Forth
- Qi

خصائص اللغة:

- لغة برمجة وظيفية.
- لغة برمجة متعددة الأساليب في البرمجة: ميتا، إجرائية.
- جامع قمامة مبني داخل اللغة.
- الاعتماد على العودية: وهو مبدأ تستمد منه ليسب قوة كبيرة خاصة في جانب الذكاء الاصطناعي.
- طريقة الكتابة ديناميكية، صارمة.
- كل شيء في ليسب قائمة.

أهم مجالات التطبيق:

- سيطرت ليسب سيطرة تامة على مجال الذكاء الاصطناعي لمدة تفوق الربع قرن وإلى الآن تعتبر أكثر لغة انتشاراً في مجال برمجة الذكاء الاصطناعي بالإضافة إلى ذلك فلغة ليسب تعتبر رائدة اللغات في مجال البرمجة الوظيفية.
- الروبوتات.
- محركات الألعاب.
- التعرف على الأنماط.
- نظم إدارة الدفاع الجوي.
- إدارة ومعالجة القوائم.
- تستخدم كلغة أكاديمية للتدريس في الجامعات (البرمجة الوظيفية).

مثال برمجي:

كود Hello world.

```
(DEFUN HELLO-WORLD ()
(PRINT (LIST 'HELLO 'WORLD))
)
```

٦. Python

تعد بايثون لغة برمجة حديثة نسبياً فبداياتها ترجع إلى سنة ١٩٩١ عندما قام جيدو فان روسام Guido van Rossum بتطويرها. أكثر خصائص بايثون مبنية ومستوحاة من لغة مفسرة تدعى ABC، حيث كانت لدى روسام رغبة في تصحيح بعض أخطاء هذه اللغة من ناحية، مع المحافظة على بعض خصائص اللغة من ناحية أخرى. في البداية كان فان روسام يبحث عن لغة مفسرة قابلة للتطويع والتوسيع تشبه ABC في طريقة كتابتها مع القدرة على استدعاء أوامر نظام Amoeba الذي كان يعمل عليه حينها. وبعد مشاورة مصممي Modula-3 قرر فان روسام البدء في مشروع لغة برمجة جديدة أسماها Python والاسم المستوحى من مسلسل كوميدي من زمن السبعينات.

لغات أثرت على لغة بايثون:

- Lisp
- Haskell
- Perl
- Java
- Icon
- ABC
- C
- Modula-3
- AlGOL 68

لغات تأثرت بلغة بايثون:

- Ruby
- Boo
- Groovy
- Cobra
- D
- Dao
- Falcon

خصائص اللغة:

- لغة مُفسرة، تفاعلية، كائنية، وظيفية.
- برامجها تعمل على جميع أنظمة التشغيل في حال توافر المفسر فقط.
- الكتابة ديناميكية، تجدر الإشارة هنا إلى وضوح وسهولة أسلوب الكتابة في بايثون.
- غنية بالإضافات والمكتبات.
- مفتوحة المصدر.

أهم مجالات التطبيق:

في الوقت الحاضر تعتبر بايثون لغة برمجة متعددة الأغراض ولكن من أكثر استعمالات بايثون:

- برمجة برامج لينكس.
- سكربتات إدارة النظام.
- التعامل مع قواعد البيانات.
- برامج الويب.

الموقع الرسمي:

https://www.python.org/

ترخيص اللغة:

Python Software Foundation License

مثال برمجي:

تخمين رقم بين ١ و ١٠.

```
import random

target, guess = random.randint(1, 10), 0
while target! guess:
guess = int(input('Guess my number between 1 and 10 until you get it right: '))
print('Thats right! )
```

٧. Fortran

تعتبر لغة فورتران واحدة من أقدم لغات البرمجة. قام بتطويرها مجموعة من المبرمجين في IBM بقيادة جون باكوس John Backus، حيث كان أول ظهور لها في سنة ١٩٥٧. وقد جاء اسم فورتران من دمج اختصار كلمتي "ترجمة الصيغ" Formula Translating، لأن الهدف الأساسي كان تسهيل عملية كتابة المعادلات الرياضية في الأكواد البرمجية.

فورتران تحتل مكانة خاصة بين لغات البرمجة لأنها تعتبر أول لغة برمجة عالية المستوى بالإضافة إلى استخدامها أول مترجم على الإطلاق. قبل ظهور فوتران كان على المبرمجين أن يبرمجوا باستخدام أسمبلي والتي كانت تحتاج إلى مجهود مُتعب في الكتابة أضف إلى ذلك عملية تصحيح الأخطاء التي كانت تحتاج مجهوداً مضاعفاً.

عليه؛ كان الهدف إنشاء لغة برمجة سهلة التعلم، مناسبة للعديد من التطبيقات، غير معتمدة على آلة معينة مع التمتع بقدرات عالية في مجال الرياضيات.

مع كل هذه المزايا استطاع المبرمجون أن يبرمجوا باستخدام فوتران ٥٠٠% أسرع من البرمجة باستخدام أسمبلي الشيء الذي أتاح فرصة أكبر للتفكير في حل المشاكل بدلاً من كتابة الأكواد وصيانتها. إذاً فورتران لها الفضل في إنشاء نظرية الترجمة في علوم الكمبيوتر.

لكن مع التطور ظهرت مشكلة تعدد إصدارات فوتران مما حدى المنظمة الأمريكية للمعايير إلى إصدار معايير موحدة في سنة ١٩٦٦ عرفت بفورتران ٦٦ لحقها إصدار فورتران ٧٧ في سنة ١٩٧٨ وإصدار فورتران ٩٠ في سنة ١٩٩٠ مع مزيد من الإضافات والمزايا لهذه اللغة العريقة.

لغات أثرت على لغة فورتران:

- Speedcoding

لغات تأثرت بلغة فورتران:

- C
- AlGOL 58
- Basic
- PL/I

خصائص اللغة:

- لغة مُترجمة وإجرائية.
- غير محصورة بجهاز معين.
- التحكم بموارد التخزين والذاكرة.
- توفر تحكم قوي للتخاطب مع الهاردوير.
- قوية جداً في التعبير عن المعادلات والتعابير والدوال الرياضية.
- كفاءة وسرعة عالية جداً لتطبيقاتها، أقل فقط ٢٠% من كفاءة برامج الأسمبلي.

أهم مجالات التطبيق:

فورتران قوية جداً في عدة مجالات، أهمها:

- برامج معالجة المعادلات الرياضية.
- البرامج الرياضية، والعلمية، والإحصائية، والهندسية.

مثال برمجي:

تخمين رقم بين ١ و ١٠.

```
program guess_the_number
implicit none

integer:: guess
real:: r
integer:: i, clock, count, n
integer,dimension(:),allocatable:: seed
real,parameter:: rmax = 10

! nitialize random number generator:
call random_seed(size=n)
allocate(seed(n))
call system_clock(count)
seed = count
call random_seed(put=seed)
deallocate(seed)

! ick a random number between 1 and rmax:
call random_number(r)! between 0.0 and 1.0
i = int((rmax-1.0)*r + 1.0)! between 1 and rmax

! et user guess:
write(*,'(A)') 'I''m thinking of a number between 1 and 10.'
do! oop until guess is correct
write(*,'(A)',advance='NO') 'Enter Guess: '
read(*,'(I5)') guess
if (guess==i) exit
write(*,*) 'Sorry, try again.'
end do

write(*,*) 'You''ve guessed my number!
end program guess_the_number
```

٨. Algol

تعتبر لغة Algol وهي اختصاراً لكلمتي"ALGOrithmic Language" واحدة من اللغات عالية المستوى المخصصة للبرمجة العلمية والحسابية. بدأت في سنة ١٩٥٠، حيث طرحت على شكل تقرير بعنوان "Algol 58" وتطورت من خلال التقارير إلى ألجول ٦٠ ثم ٦٨.

صُممت اللغة من قبل لجنة عالمية لكي تصبح لغة عالية المستوى. وقد طُرحت مشكلة الانتقالية في تطوير البرامج من خلال أول اجتماع للجنة في مدينة زيورخ شمال سويسرا. وقد جعلت خاصية الانتقالية وعدم الاعتماد على آلة معينة، المصممين أكثر إبداعاً ولكن في نفس الوقت جعلت عملية التطبيق أكثر صعوبة.

وعلى الرغم من عدم وصول Algol إلى مستوى عالٍ من الشعبية التجارية كما حصل للغتي فورتران وكوبول، إلا إنها تعتبر أهم لغة في عصرها من ناحية تأثيرها القوي على اللغات القادمة. نظام المفردات والهيكل النحوي الخاص بها أصبح شديد الشهرة لدرجة أنه فعلياً جُل لغات البرمجة يقال عنها "مشابهة للغة Algol".

لغات تأثرت بلغة ألجول:

- C
- C++
- Ada
- Python
- SteelMan
- Bash
- Simula
- Pascal

خصائص اللغة:

- لغة مُترجمة.
- أسلوب الكتابة ثابت.
- لغة متعددة النماذج، أمرية، متزامنة.

أهم مجالات التطبيق:

- أهم مجال للغة Algol كان استخدامها للأبحاث العلمية والحسابات بواسطة العلماء في أوروبا وأمريكا. ولكن على المستوى التجاري لم يكتب لها النجاح لأسباب عديدة من أهمها عدم اهتمام الشركات الكبيرة باللغة.

مثال برمجي:

برنامج تخمين رقم بين ١ و ١٠.

```
main:
(
INT n;
INT g;
n:= ENTIER (random*10+1);
PROC puts = (STRING string)VOID: putf(standout, ($gl$،string));
puts("I'm thinking of a number between 1 and 10.");
puts("Try to guess it! ");
DO
readf(($g$، g));
IF g = n THEN break
ELSE
puts("That's not my number. ");
puts("Try another guess! )
FI
OD;
break:
puts("You have won! ")
)
```

٩. COBOL

تعتبر لغة COBOL واحدة من أوائل لغات البرمجة عالية المستوى (وهي اختصار لجملة COmmon Business-Oriented Language). طُورت في سنة ١٩٥٩ من قبل مجموعة من محترفي الكمبيوتر ومنذ ذلك الحين خضعت اللغة للعديد من التعديلات والتحسينات.

ولحل مشكلة عدم التوافقية بين إصدارات كوبول المتعددة قامت المنظمة الوطنية الأمريكية للقياسات بإصدار نسخة موحدة للغة في سنة ١٩٦٨، حيث عرف هذا الإصدار باسم ANSI COBOL. ومع سنة ١٩٧٤ قامت المنظمة مرة أخرى بإعادة طرح نسخة معدلة من كوبول تحتوي على المزيد من المزايا والإضافات الجديدة. وتكررت هذه العملية أيضاً في سنة ١٩٨٥.

وفي الإصدارة الرابعة التي عرفت باسم كوبول ٩٧ أُضيفت خصائص البرمجة الكائنية. ويوجد الآن العديد من المترجمات لكوبول على الرغم من محاولات التوحيد في هذا المجال.

لغات أثرت على لغة كوبول:

- FACT
- COMTRAN
- FLOW-MATIC

لغات تأثرت بلغة كوبول:

- PL/I
- COBOL Script
- ABAP

خصائص اللغة:

- لغة مُترجمة.
- مخصصة لإدارة الأعمال للشركات.
- إمكانية التمازج مع تطبيقات الويب.
- بيئة تطوير مرئية.
- التفاصيل في تعريف المتغيرات مثل عدد كسور المتغير وموضع نقطة الكسر.
- تعنى بتفاصيل الملفات ومعلوماتها مما يجعلها خيارا ممتازا لطباعة التقارير.
- توافر مكتبات تحوي العديد من الكلاسات.
- نمط الكتابة صارم.

أهم مجالات التطبيق:

من اسم اللغة يظهر أنها خيار ممتاز لحل مشاكل الأعمال والشركات، حيث تستخدم كثيراً في الشركات كنظام شامل خاصة في تتبع المصادر والمخارج وغير ذلك.

مثال برمجي:

برنامج تخمين رقم بين ١ و ١٠.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Guess-The-Number.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 Random-Num PIC 99.
01 Guess PIC 99.

PROCEDURE DIVISION.
COMPUTE Random-Num = 1 + (FUNCTION RANDOM * 10)
DISPLAY "Guess a number between 1 and 10:"

PERFORM FOREVER
ACCEPT Guess

IF Guess = Random-Num
DISPLAY "Well guessed!
EXIT PERFORM
ELSE
DISPLAY "That isn't it. Try again."
END-IF
END-PERFORM

GOBACK
```

١٠. PHP

في البداية ظهرت لغة PHP لأول مرة باسم PHP/FI وقد أسسها راسموس ليردورف Rasmus Lerdorf في سنة ١٩٩٥ على شكل مجموعة من سكربتات مكتوبة بلغة البيرل لكي يسيطر على إحصائيات موقعه ويكسب بعض المعلومات عن رواد موقعه، وما لبث أن أطلق عليها اسم Personal Hompage Tools أي "أدوات تصميم الصفحات الشخصية".

طبعاً للتوسع في تغطية بعض العمليات الإضافية بدأ راسموس بصنع آلية بلغة سي بوسعها الاتصال بقواعد البيانات، وتمكن المستخدمين أن يصنعوا لهم صفحات ديناميكية بسيطة. وأخيراً قرر راسموس أن يعرض هذا الكود المصدري المكتوب بلغة سي على الجمهور لكي يستطيع أي شخص استخدامه أو حتى تصليح بعض الأخطاء التي قد توجد حتى أن بعضهم عمل على توسيع الكود بإضافة بعض الخصائص الجديدة. طبعاً في هذه المرحلة كانت بي اتش بي تحتوي على دوال Functions أقل بكثير مما نعرفها الآن وكان بها بعض الشبه من لغة بيرل ولكن طبعاً بإمكانيات أكثر تواضعاً من أن تقارن بلغة بيرل.

في سنة ١٩٩٧، أُصدرت النسخة الثانية من PHP/FI والتي كانت تحتوي على النسخة الجديدة الثانية من الكود المصدري المكتوب بلغة سي، وكان هناك الآلاف من المستخدمين يستخدمونها وحوالي ٥٠٠٠٠ موقع أعلن تنصيبه لبي اتش بي على خوادمه.

وفي سنة ١٩٩٨ كانت بي اتش بي قد اكتسبت قاعدة جماهيرية كبيرة ومئات من الآلاف من المواقع كانت ترسل معلنة أنها قامت بتنصيب بي اتش بي على خوادمها. حتى بلغت نسبة المواقع التي تستخدم بي اتش بي ٣ حوالي ١٠٪ من إجمالي مواقع الويب.

وقد صدرت PHP 3 رسمياً في شهر يونيو من سنة ١٩٩٨ بعد أن أمضت حوالي تسعة أشهر تحت الاستخدام التجريبي.

وفي شتاء ١٩٩٨ وبعد فترة وجيزة من إصدار PHP 3 الرسمي. بدأ زيف سوراسكي وآندي جوتمانز Andi Gutmans بإعادة كتابة وبرمجة نواة بي اتش بي.

لهذا الغرض بدأ تصميم محرك جديد سمي "ZEND Engine" (و يتركب اسم المحرك الجديد من أول حرفين من اسم زيف وآخر حرفين من اسم آندي). وفي منتصفات ١٩٩٩ تم التعريف بهذا المحرك لأول مرة بعد أن حقق الأهداف المنشودة منه بنجاح قوي وفي شهر مايو من سنة ٢٠٠٠ صدرت PHP 4 رسمياً.

وفي سنة ٢٠٠٤ صدرت PHP 5 مع المحرك الثاني ونظام كائني جديد مما قدم قدرات كائنية جديدة وقوية.

لغات أثرت على لغة بي اتش بي:

- Perl
- C
- Java
- C++

لغات تأثرت بلغة بي اتش بي:

- PHP4Delphi
- Falcon

خصائص اللغة:

- لغة مُفسرة.
- مخصصة لتطوير الويب.
- تعمل على أغلب نظم التشغيل.
- كودها يعمل داخل وسوم HTML.
- غنية بدوال كثيرة مضمنة داخل اللغة.
- أسلوب كتابة ديناميكي.
- سهلة التعلم.

أهم مجالات التطبيق:

- بلا شك أهم مجال لبي اتش بي هو قوتها في مجال تطوير مواقع وتطبيقات الويب.
- يمكن أيضاً إنشاء برامج ذات واجهة رسومية خاصة بسطح المكتب.

موقع اللغة:

http://www.php.net/

ترخيص اللغة:

PHP License

مثال برمجي:

برنامج تخمين رقم بين ١ و ١٠.

```
<?php

session_start();

if(isset($_SESSION['number']))
{
$number = $_SESSION['number'];
}
else
{
$_SESSION['number'] = rand(1,10);
}

if($_POST["guess"]){
$guess = htmlspecialchars($_POST['guess']);

echo $guess. "<br />";
if ($guess! $number)
{
echo "Your guess is not correct";
}
elseif($guess == $number)
{
echo "You got the correct number! ;
}


}
?>
```

```
<! OCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-
8" />
<title>Guess A Number</title>
</head>
<body>
<form action="<?=$_SERVER['PHP_SELF'] ?>" method="post"
name="guess-a-number">
<label for="guess">Guess number:</label><br/ >
<input type="text" name="guess" />
<input name="number" type="hidden" value="<?= $number ?>" />
<input name="submit" type="submit" />
</form>
</body>
</html>
```

١١. Eiffel

لغة إيفل من إنشاء برتنارد مير Bertrand Meyer ومن تطوير شركته "هندسة البرامج التفاعلية"، بدأ العمل عليها في ١٩٨٥ وكان أول ظهور لها في سنة ١٩٨٦. وسميت ب إيفل تيمنا بجوستاف إيفل المهندس الذي صمم برج إيفل المشهور. ويضيف مطوري هذه اللغة أنه باستخدامك هذه اللغة في مشاريعك ستتمكن من إنجاز المشاريع في وقتها وضمن الإطار المالي المرصود كما حدث تماماً في بناء برج إيفل!

إيفل لغة تُعنى بالجودة والكفاءة وقابلية الاستفادة من الكود في مشاريع لاحقة، إضافة إلى أنها قدمت مبادئ برمجية جديدة وجدت طريقها لاحقاً إلى لغات البرمجة المشهورة اليوم مثل جافا.

لغات أثرت على لغة إيفل:

- Ada
- Simula
- Z

لغات تأثرت بلغة إيفل:

- Java
- C#
- Ruby
- D
- Lisaac
- Sather

خصائص اللغة:

- لغة مُترجمة.
- لغة برمجة كائنية.
- تعمل على أغلب أنظمة التشغيل.
- تدعم التكرار والتعدد في التوارث.
- أسلوب الكتابة ثابت، صارم.
- التوثيق الآلي.
- البرمجة بالعقود Design by contract.

أهم مجالات التطبيق:

- أنظمة الاتصالات.
- التدريس الأكاديمي.
- النمذجة الأولية السريعة.
- البرامج التجارية.
- برمجة الألعاب.
- البرامج الطبية.
- برامج الطيران.

موقع اللغة:

http://www.eiffel.com/

مثال برمجي:

كود Hello world.

```
class
HELLO_WORLD
create
make
feature
make
do
print ("Hello, world! N")
end
end
```

١٢. Modula-2

في منتصف السبعينات، كان نيكلاوس ويرث Niklaus Wirth (مصمم لغة باسكال) يقوم بتجارب ودراسات بالتزامن أدت إلى إنشاء لغة جديدة اسمها مودولا. ولكن مودولا لم تصدر بشكل رسمي أبداً حيث وقف تطويرها بعد نشر تقريرها. ولكن نيكلوس قام ببناء لغة برمجة جديدة هدفها الأساسي هو أن تكون لغة خاصة بجهاز سيعرف باسم Lilith.

طبعاً فشل الجهاز ولم يحقق النجاح المطلوب ولكن لغته الخاصة هذه نُشرت في سنة ١٩٨٠ لنعرفها الآن باسم مودولا-٢. هذه اللغة على بساطتها إلا أنها قوية وجبارة بحيث كانت اللغة المنتشرة في أوروبا حتى مع موجود جافا وسي بلس بلس لاحقاً. وقد اعتبرها مبرمجها كخليفة لغة البرمجة باسكال.

لغات أثرت على لغة مودولا:

- Pascal
- ALGOL
- Mesa
- Simula-67

لغات تأثرت بلغة مودولا:

- Ada
- Oberon
- Lua
- Fortran 90
- Modula-3
- Modula-GM

خصائص اللغة:

- لغة مُترجمة.
- تعمل على أغلب أنظمة التشغيل.
- أسلوب الكتابة ثابت، صارم.
- لغة برمجة أمرية (إلزامية)، تنظيمية، تعتمد الوحدات (مودلر).
- ميزة الوحدات أعطتها قوة كبيرة في تطوير المشاريع العملاقة.
- القدرة على البرمجة العالية والمنخفضة المستوى.
- تعتبر سهلة التعلم بسبب صغر قاموسها النحوي.

أهم مجالات التطبيق:

- برمجة الأنظمة.
- البرمجة المتزامنة.
- برمجة الأنظمة المضمنة (المدمجة).
- هندسة البرامج.
- التعليم.
- البرمجة الصوتية.

مثال برمجي:

كود Hello world.

```
MODULE hello;
FROM InOut IMPORT writestring, writeln;
begin
WriteString("Hello, world! );
Writeln;
end hello.
```

١٣. Ruby

بدأت فكرة لغة البرمجة روبي في سنة ١٩٩٣ عندما أراد يوكيهيرو ماتسوموتو Yukihiro Matsumoto (أو كما يحب أن يعرف Matz) أن يطور لغة برمجة تتفوق على بيرل في القوة وتكون أكثر كائنية من بايثون، لغة تجمع بين الوظيفية والأمرية. اختير اسم روبي للغة قبل البدء في كتابة اللغة حيث كان هناك أيضاً اسم كورال مطروحاً ولكن استبعد الأخير لوجود لغة برمجة أخرى بهذا الاسم.

في يوم ٢١ ديسمبر من سنة ١٩٩٥ صدرت روبي ٠,٩٥ للعامة تلاه ثلاث إصدارات متلاحقة خلال يومين فقط. رافق هذا الإصدار الإعلان عن انطلاق القائمة البريدية روبي باللغة اليابانية. في ديسمبر ٢٥ من سنة ١٩٩٦ خرج الإصدار ١.٠ من لغة روبي ثم تلاه الإصدار ١.٣ في سنة ١٩٩٩ حيث رافقته انطلاقة القائمة البريدية الإنجليزية.

هذا الأمر تسبب في ازدياد شعبية اللغة وصدر أول كتاب انجليزي لهذه اللغة في سنة ٢٠٠٠ باسم "برمجة روبي" Ruby Programming تجدر الإشارة هنا إلى أن هذا الكتاب قد طُرح مجاناً لاحقاً. صدرت روبي ١.٩.١ في يناير ٣٠ من سنة ٢٠٠٩، ومؤخراً أطلق إصدار روبي ٢.١ في يوم الكريسماس ٢٥ ديسمبر ٢٠١٣.

لغات أثرت على لغة روبي:

- Perl
- SmallTalk
- Lisp
- Python
- Eiffel

لغات تأثرت بلغة روبي:

- Groovy
- Falcon
- Nu
- Loke

خصائص اللغة:

- لغة مفسرة.
- لغة متعددة النماذج، أمري، وظيفي، كائني.
- أسلوب الكتابة ديناميكي.
- مفتوحة المصدر.
- لديها جامع قمامة Garbage Collection.
- تعمل على منصة جافا باستخدام Jruby.
- تعمل على أغلب أنظمة التشغيل.
- سهولة كتابة الإضافات بلغة سي.

أهم مجالات التطبيق:

- روبي لغة برمجة عامة متعددة الأغراض ولكن يبدو أن من أهم استخداماتها الآن برمجة تطبيقات الويب باستخدام Ruby on Rails. أيضاً تستخدم روبي في أغراض أخرى مثل التجسيم ثلاثي الأبعاد والمحاكاة وإدارة النظم.

موقع اللغة:

https://www.ruby-lang.org/en/

ترخيص اللغة:

Ruby License أو BSD License

مثال برمجي:

تخمين الرقم بين ١ و ١٠.

```
n = rand(10) + 1
puts 'Guess the number: '
puts 'Wrong! Guess again: ' until gets.to_i == n
puts 'Well guessed!'
```

١٤. Pascal

طُورت لغة البرمجة Pascal أساساً من قِبَل "نيكولاس ورث" Niklaus Wirth وهو عضو الفدرالية العالمية لمعالجة النصوص IFIP. قام البرفسور نيكولاس ورث بتطوير باسكال لتحتوي المميزات التي تخلو منها لغات البرمجة في ذلك الوقت. وكان هدفه الرئيسي في أن يجعل من لغة باسكال:

١. لغة ذات كفاءة في مرحلتي التطبيق والتنفيذ.
٢. لغة تسمح بتطوير برامج ذات هيكلية جيدة وتنظيم رشيق.
٣. لغة لغرض تعليم مبادئ البرمجة الأساسية والمهمة.

وتعتبر لغة باسكال والتي سميت تيمناً بعالم الرياضيات "بليز باسكال" Blaise Pascal وريثة مباشرة للغة البرمجة ALGOL60 والتي بدورها هي الأخرى حظيت بكون البرفسور ورث أحد مطوريها. أيضاً فلغة باسكال تبني على العناصر البرمجية للغتي Algol w و ALGOL68. وقد ظهر أول تعريف للغة باسكال في سنة ١٩٦١، تلاه إعادة تصحيح في سنة ١٩٧٣.

وقد صُممت لكي تكون اللغة المستخدمة في الكليات لتعليم البرمجة ومفاهيم البرمجة وقد كانت بالفعل هي اللغة المفضلة في هذا المجال من نهايات الستينات إلى بداية التسعينات. وهنا بعض مميزات اللغة لمجال تعليم البرمجة الهيكلية:

- احتواؤها على Data types وهي أنواع البيانات الموجودة سلفاً في اللغة مثل الأعداد الصحيحة والمنطقية والحروف ...إلخ.
- إضافة إلى ذلك يمكن إنشاء أنواع بيانات جديدة يحددها المستخدم.
- احتواؤها على مجموعة جيدة من قوالب البيانات المهيكلة مثل: القوائم والريكوردز Sets.
- استخدام البرامج الضمنية أو ما يعرف بال Procedures and Functions.

لغات أثرت على لغة باسكال:

- ALGOL
- COBOL

لغات تأثرت بلغة باسكال:

- Java
- Oberon
- Oberon-2
- Ada
- Oxygene
- Modula-2
- Component Pascal
- Object Pascal

خصائص اللغة:

- النموذج أمري، هيكلي، إجرائي.
- لغة مترجمة، أغلب المترجمات كتبت بلغة باسكال نفسها ولكن جنو باسكال مكتوب بلغة سي.
- أسلوب الكتابة قوي وصارم.
- تدعم الـ Pointers.
- يوجد لها مفسر أيضاً.

أهم مجالات التطبيق:

- كما اشرنا سابقاً فالمجال الأهم لباسكال هو بيئة التعليم. هذه اللغة أساساً لم تصمم إلا لهذا الغرض وهي إلى الآن خيار جيد لتعليم البرمجة الهيكلية ولكن ما جعلها تفقد مكانها في بداية التسعينات هو ازدياد شعبية لغات البرمجة الشيئية مثل C++ وجافا وسمول توك وبدء الجامعات والكليات تدريس مفاهيم هذه النوعية من البرمجة. ولكن تجدر الإشارة إلى أنه للمهتمين بالبرمجة الشيئية يوجد أوبجكت باسكال وقد برمج برنامج Skype الشهير بها.

مثال برمجي:

تخمين الرقم بين ١ و ١٠.

```
Program GuessTheNumber(input, output);

var
number, guess: integer;

begin
randomize;
number:= random(10) + 1;
writeln ('I''m thinking of a number between 1 and 10, which you should guess.');
write ('Enter your guess: ');
readln (guess);
while guess <> number do
begin
writeln ('Sorry, but your guess is wrong. Please try again.');
write ('Enter your new guess: ');
readln (guess);
end;
writeln ('You made an excellent guess. Thank you and have a nice day.');
end.
```

١٥. Pl/1

طُورت لغة البرمجة Pl/1 بواسطة IBM في أواسط الستينات من القرن العشرين. وقد كان الاسم الأصلي للغة NPL (لغة البرمجة الجديدة) ولكن تم تغيير الاسم إلى Pl/1 لتفادي المغالطة بين اسمها وبين National Physical Laboratory (مختبر الفيزياء الوطني) في انجلترا. لذا لو طُور المترجم خارج انجلترا لربما بقي الاسم بدون تغيير.

قبل تطوير هذه اللغة البرمجية كانت لغات البرمجة تركز على جانب معين من التطبيق مثلاً الذكاء الاصطناعي أو الحسابات الرياضية أو المشاريع التجارية. ولكن Pl/1 لم تصمم لكي تستخدم بهذه الطريقة، بل كانت أول لغة برمجة ضخمة هدفها أن تغطي أغلب مجالات التطبيق. لذلك كان على Pl/1 أن تواجه تحديات قوية لتنافس فورتران في المجال العلمي وكوبول في مجال الأعمال، فكان من هذه الأهداف والتحديات:

١. أن تكون لغة مترجمة مع سرعة تنفيذ مشابهة لفورتران.

٢. أن تكون قابلة للتوسيع لدعم المزيد من العتاد والأجهزة والتطبيقات الجديدة.

٣. زيادة الإنتاجية وتحسين الوقت المحتاج لعملية البرمجة من خلال نقل المجهود من المبرمج إلى المترجم.

٤. أن تكون متعددة المنصات وان تعمل بشكل جيد على مستوى كل قطع العتاد ونظم التشغيل.

لغات أثرت على لغة بي إل ون:

- COBOL
- Fortran
- ALGOL

لغات تأثرت بلغة بي إل ون:

- SP/k
- REXX

خصائص اللغة:

- لغة مُترجمة.
- النموذج أمري، هيكلي، إجرائي.
- تدعم العودية، ومفاهيم البرمجة الهيكلية.
- تحاول أن تحاكي اللغة الإنجليزية في طريقة الكتابة.
- لا توجد بها كلمات مفتاحية محجوزة (يمكن أن يكون اسم المتغير كلمة مفتاحية).

أهم مجالات التطبيق:

- لغة Pl/1 كانت تتمتع بشعبية كبيرة في جانب إدارة الأعمال والتطبيقات العلمية. وكانت اللغة الرئيسية للتدريس في جامعة ميشيغان ديربورن لمدة من الزمن. أما في الوقت الحاضر فشعبيتها قلت كثيراً طبعاً (و لكنها لا تزال تستخدم) بسبب اللغات الجديدة والمفاهيم الحديثة. ومن الشركات التي استخدمت Pl/1 شركة فورد Ford المعروفة وشركة مارثون النفطية Marathon.

مثال برمجي:

تطبيق Hello World.

```
Hello2: proc options(main);
put list ('Hello, world! );
end Hello2;
```

١٦. Haskell

بعد إصدار لغة البرمجة Miranda في سنة ١٩٨٥، زاد الاهتمام بلغات البرمجة الوظيفية الكسولة. فمع سنة ١٩٨٧ ارتفع عدد اللغات الوظيفية الصرفة إلى أكثر من ١٢ لغة. طبعاً من هذه اللغات كانت ميرندا Miranda الأكثر استخداماً ولكنها لم تكن مجانية بل مملوكة لشركة Research Software. ولهذا السبب في مؤتمر اللغات البرمجية الوظيفية وهندسة الكمبيوتر (FPCA 87) والذي كان في Portland Oregon، عُقد اجتماع أبدى فيه المشاركون ضرورة إنشاء لجنة لتعمل على إنشاء معايير مفتوحة لهذه اللغات.

ومع سنة ١٩٩٠ تم الانتهاء من تعريف هاسكل ١.٠ وفي سنة ١٩٩٧ توجت الجهود بظهور هاسكل ٩٨ التي وفرت إصداراً ثابتاً، خفيفاً ومتنقلاً من اللغة، بالإضافة إلى مكتبة لغرض التعليم. وقد رحبت اللجنة بإنشاء الإضافات والبدائل لهاسكل ٩٨ عن طريق إضافة الخصائص التجريبية.

لغة هاسكل تتطور بشكل سريع جداً ويعتبر المترجم "جلاسكو" GHC هو الأكثر شيوعاً في الاستخدام. يذكر أن سبب التسمية هو تيمناً بعالم الرياضيات والمنطقي الأمريكي هاسكل كوري Haskell Curry.

لغات أثرت على لغة هاسكل:

- Lisp
- Miranda
- APL
- Ponder
- Standard ML
- Lazy ML

لغات تأثرت بلغة هاسكل:

- Omega
- Perl6
- Python
- Visual Basic 9
- Cayenne
- Java Generics
- F#
- C#
- Scala

خصائص اللغة:

- لغة برمجة وظيفية صرفة.
- أسلوب الكتابة ثابت وصارم.
- ذات معايير ومواصفات مفتوحة.
- تستخدم التقييم الكسول Lazy Evaluation .
- تتمتع بمجتمع فاعل ولديها مخزن غني بالمكتبات يدعى Hackage.
- مترجم GHC يعتبر مترجماً ومفسراً أيضاً ويعمل على أغلب نظم الأنظمة ويتميز بالكفاءة العالية.

أهم مجالات التطبيق:

- هاسكل بدأت تزداد شعبيتها في الاستخدامات التجارية. المبرمجة المشهورة أودري تانج قامت بعمل تطبيق لبيرل ٦ بلغة هاسكل فكانت النتيجة نسخة تعمل بشكل جيد في زمن قياسي معروفة باسم Pugs. أيضاً توزيعة Linspire اختارت هاسكل كلغة تطوير لأدوات النظام. Xmonad وهو مدير نوافذ لنظام نوافذ X11 كُتب كاملاً باستخدام هاسكل. وهناك الكثير من الاستخدامات والبرامج لهاسكل على صعيد التطوير وقواعد البيانات وحتى المترجمات والألعاب.

موقع اللغة:

http://www.haskell.org/

مثال برمجي:

تخمين الرقم بين ١ و ١٠.

```
import Control.Monad
import System.Random

-- Repeat the action until the predicate is true.
until_ act pred = act >>= pred >>= flip unless (until_ act pred)

answerIs ans guess
| ans == guess = putStrLn "You got it! >> return True
| otherwise = putStrLn "Nope. Guess again." >> return False

ask = liftM read getLine

main = do
ans <- randomRIO (1,10):: IO Int
putStrLn "Try to guess my secret number between 1 and 10."
ask `until_` answerIs ans
```

١٧. Visual Basic

في سنة ١٩٩١ عُرض الإصدار ١.٠ من لغة فيجوال بيسك، والتي قدمت طريقة السحب والإفلات لتصميم واجهات المستخدم، والتي طُورت من خلال برنامج إنشاء النماذج Forms الذي أنشأه الآن كوبر Alan Cooper وشركته المعروف باسم Tripod.

حيث وقع تعاقد بين ميكروسوفت وكوبر (و شركاؤه) لتطوير Tripod ليكون فورم سيستم قابل للبرمجة لويندوز ٣.٠، وذلك تحت المسمى البرمجي Ruby (لا توجد هنا أي علاقة مع لغة البرمجة روبي). على الجانب الآخر Tripod لم تكن تحتوي على لغة برمجة إطلاقاً، لذلك قررت ميكروسوفت بأن تدمج روبي مع لغة البرمجة Basic لتنشئ ما يعرف بفيجوال بيسك (بيسك المرئية: لتركيزها على الجانب المرئي في إنشاء البرامج بسرعة).

في نوفمبر سنة ١٩٩٢ صدرت VB 2.0 حيث تم تحسين بيئة البرمجة لتكون أكثر سهولة وأكثر سرعة. وفي صيف سنة ١٩٩٣ صدرت VB 3 بنسختيها القياسية والاحترافية. وفي هذا الإصدار أُضيف الإصدار ١.١ من Microsoft Jet Database Engine. وفي سنة ١٩٩٥ صدرت VB 4 هذا الإصدار كان الأول في دعم إنشاء برامج 16bit و 32bit. أيضاً مع هذا الإصدار أصبحت هناك إمكانية إنشاء كلاسات غير ذات واجهة رسومية. وقد عانى هذا الإصدار بعض المشاكل في التوافقية.

مع الإصدار الخامس في فبراير ١٩٩٧ قررت ميكروسوفت أن تصدر VB حصرياً لمنصة 32bit. في هذا الإصدار أصبح هناك إمكانية إنشاء أزرار من تصميم المستخدم بالإضافة إلى القدرة لبناء البرامج مباشرة إلى الكود البرمجي التنفيذي لويندوز. مع سنة ١٩٩٨، صدرت النسخة السادسة من فيجوال بيسك مع العديد من التحسينات أهمها القدرة على إنشاء برامج الويب. وقد قامت ميكروسوفت في سنة ٢٠٠٨ بإلغاء VB 6.

ثم أتى بعد ذلك VB.NET الذي يعد وريث VB6، والذي هو جزء من منصة NET. ولا يوفر أي توافقية مع الإصدارات السابقة من VB على الرغم من وجود برامج تقوم بالتحويل بين أكواد الإصدارين إلا أن التحويل الأوتوماتيكي الكامل غير ممكن لأغلب المشاريع. تجدر الإشارة إلى أنه إلى الآن هناك مجتمع كبير من المستخدمين لا يزالون يدعمون ويبرمجون بالإصدار السادس من VB.

لغات أثرت على لغة فيجوال بيسك:

- QuickBASIC

لغات تأثرت بلغة فيجوال بيسك:

- VisualBasic.NET
- REALBasic
- Gambas
- Basic4ppc

خصائص اللغة:

- كائنية التوجه.
- أسلوب الكتابة ثابت وصارم.
- تعمد على Event Driven.
- لديها جامع قمامة.
- بشكل عام ليست حساسة لحالة الحروف Case-insensitive.
- اندماج قوي مع نظام التشغيل ويندوز.
- يوجد لها مترجم من الإصدار الخامس جنباً إلى جنب مع المفسر.

بعض عيوب فيجوال بيسك حتى الإصدار السادس (قبل إصدار دوت نت):

- دعم ضعيف للبرمجة الكائنية.
- الاعتمادية على تعقيدات مدخلات الريجستري لـ COM.
- قبل الإصدار الخامس كانت هناك مشاكل من ناحية الأداء للبرامج المكتوبة بهذه اللغة، أُزيلت مع الإصدار الخامس.
- مشاكل التوافقية بسبب تعدد الإصدارات.

أهم الاستخدامات:

فيجوال بيسك كما لغة البرمجة BASIC هدفها الأساسي هو تسهيل عملية البرمجة، حيث أنها وفرت خاصية البرمجة السريعة باستخدام السحب والإفلات للعناصر لكي يتم إنشاء الواجهة الرسومية بيسر وسهولة، هذا إضافة إلى توفير قيم افتراضية لأغلب العناصر ما يساعد في تقليل كتابة الأكواد من قبل المبرمج فأصبح إنشاء البرامج لويندوز شيء سهل وكأنه مجرد تصميم لصحفة ويب هذا بالطبع لا يعني أنه لا توجد هناك إمكانية لإنشاء برامج عملاقة ومعقدة باستخدام فيجوال بيسك. عليه فان أغلب استخدامات فيجوال بيسك تقع في برامج ويندوز سواء الصغيرة أو الكبيرة.

صفحة اللغة:

http://msdn.microsoft.com/en-us/vstudio/hh388573.aspx

مثال برمجي:

```
Private Sub Form_Load()
‘ execute a simple message box that will say "Hello, World!
MsgBox "Hello, World!
End Sub
```

١٨. JavaScript

طُورت جافا سكربت بواسطة براندن ايك Brenden Eich من شركة Netscape تحت مسمى Mocha، والذي تم تغييره لاحقاً إلى LiveScript وفي النهاية إلى جافا سكربت. أُضيفت وطُبقت جافا سكربت للمرة الأولى في الإصدار 2.0B3 من المتصفح العريق نتسكيب ديسمبر ١٩٩٥.

وقد تسبب اسم "جافا سكربت" إلى نوع من المغالطة في الربط بينها وبين لغة جافا المشهورة وأعزى البعض ذلك أنه حركة تسويقية متعمدة من نتسكيب. في حقيقة الأمر جافا سكربت ليس لها علاقة بلغة جافا من شركة صن Sun ولكن الصفات المشتركة بين اللغتين كثيرة.

يظهر هذا التشابه جلياً في طريقة الكتابة والتي هي مشابهة للغة البرمجة سي، أضف إلى ذلك أن جافا سكربت تتبع قواعد التسمية على طريقة جافا. وقد قيل أن سر التسمية يكمن في صفقة بين نتسكيب وصن حيث تقوم نتسكيب بإضافة بيئة تشغيل جافا في متصفحها الذائع الصيت وقتها. جافا سكربت تعتبر علامة مسجلة لشركة صن (أوركل حالياً) وقد استخدمت تحت ترخيص لتقنيات مطورة من نتسكيب وموزيلا Mozilla.

المبادئ الأساسية في تصميم اللغة اقتبست من لغتي Self و Scheme. وبسبب انتشار ونجاح جافا سكربت كلغة تعمل ناحية العميل في مواقع الويب قامت ميكروسوفت بإنشاء إصدارة متوافقة خاصة بها اسمتها JScript لتتفادى مسائل الترخيص. وقد أُضيفت JScript في الإصدار الثالث من إنترنت إكسبلورر.

قامت نتسكيب بتقديم جافا إلى Ecma International بهدف توحيد المعايير مما أدى إلى ظهور المعيار الموحد ECMAScript.

وقد أصبحت جافا سكربت واحدة من اللغات الشعبية جداً في برمجة مواقع الويب، على الرغم من ذلك فكثير من المبرمجين المحترفين تجنبوا هذه اللغة بسبب أن الشريحة المستهدفة هي فئة مصممي مواقع الويب والهواة وغير ذلك من الأسباب.

لكن مع تطور تقنية Ajax عادت جافا سكربت للأضواء مجدداً مع إضافة برمجة احترافية جديدة. وكانت النتيجة التحصل على عدد كبير من المكتبات و Frameworks، مما أثر إيجابياً على تحسن مشاريع البرمجة وزيادة استخدام جافا سكربت خارج إطار المتصفح.

لغات أثرت على لغة جافا سكربت:

- Scheme
- Self
- Perl
- Python
- Java
- C

لغات تأثرت بلغة جافا سكربت:

- Objective-J
- Jscript
- Jscript.NET

خصائص اللغة:

- متعددة النماذج: وظيفية، كائنية.
- لغة نصية (سكربت)
- غير معتمدة على نظام تشغيل محدد
- أسلوب الكتابة ديناميكي
- تستخدم ال Prototypes بدلاً من الكلاسات للوراثة.
- لديها قدرة تعامل قوية مع التعابير النمطية على طريقة لغة البرمجة بيرل.
- تحتاج إلى محرك يقوم بتفسير الأكواد المصدرية، ويعتبر سبايدر مونكي أول محرك لجافا سكربت.
- إمكانية تضمينها داخل صفحات HTML.

أهم مجالات التطبيق:

- طبعاً الاستخدام المبدئي لجافا سكربت هو في مواقع الويب ولكن هناك بعض من البرامج قامت بإضافة أو تضمين مفسر جافا سكربت. من هذه البرامج التي نفذت بجافا سكربت:
  - Apple Dashboard Widgets.
  - Microsoft Gadgets.
  - Yahoo! Widgets.
  - Google Desktop Gadgets.
- والكثير من البرامج الأخرى التي تضيف دعم للسكربتنج من خلال جافا سكربت مثل أدوبي اكروبات وفوتوشوب ودريم ويفر وأوبن أوفس ...إلخ.
- ومؤخرا بدأت تنافس لغات مثل PHP للبرمجة من جهة الخوادم Server side من خلال Node.js.

مثال برمجي:

```
<script type="text/javascript">
<! to hide script contents from old browsers
document.write("Hello World! )
// end hiding contents from old browsers ->
</script>
```

١٩. C++

بدأ بيارن ستروستروب Bjarne Stroustrup العمل على مشروع "C with classes" في سنة ١٩٧٩، وذلك بعد أن جاءته فكرت عمل لغة برمجة جديدة بعد الخبرة التي اكتسبها في البرمجة لرسالة الدكتوراه. وقد كان ستروستروب معجباً بلغة البرمجة سمولا Simula لأنها كانت تحتوي على كثير من المزايا المناسبة والمساعدة لبناء مشاريع برمجية عملاقة ولكن ما يعيب هذه اللغة هو كونها بطيئة مما يجعلها غير عملية في أرض الواقع.

في الجهة الأخرى كانت لغة BCPL سريعة جداً ولكن يعيبها أنها منخفضة المستوى بشكل يجعلها غير مناسبة لتطوير المشاريع العملاقة. بناء على ذلك قرر ستروستروب أن يطور لغة سي بإضافة مزايا سيمولا. وقد اختار لغة سي لأنها سريعة، متعددة الأغراض، متنقلة، وذات شعبية واسعة. ولم تكن سيمولا وحدها من أثرت على سي بلس بلس بل هناك المزيد من اللغات مثل ألجول ٦٨ و CLU. فمن ضمن أولى الإضافات نذكر: الكلاسات، المرسلات الافتراضية.

في سنة ١٩٨٣ أُعيد تسمية اللغة إلى C++، وتلاها إضافات جديدة للغة نذكر منها: الثوابت، تعليقات السطر الواحد، المراجع.

وفي سنة ١٩٨٥ صدر كتاب The C++ Programming Language في نسخته الأولى، ليكون من أهم المصادر للغة لعدم توافر معيار موحد وقتها. في سنة ١٩٨٩ صدرت C++ 2.0 مع مزيد من الإضافات مثل: تعدد التوارث، الكلاسات المجردة.

فـي سـنة ١٩٩٠ صـدر The annotated C++ Reference Manual، والـذي أصـبح القاعـدة لإنشـاء المعيـار فـي المسـتقبل. ومـن الإضـافات المتـأخرة نـذكر: القـوالب، الاسـتثناءات، المساحات. ومع تطور لغة سي بلس بلس تطورت معها مكتبة قياسـية وقد كانت أولـى المكتبات إضافة هي مكتبة I/O Stream، ومن أهم المكتبات الآن هي مكتبة القالب القياسي.

ولا تزال سي بلس بلس تحظى بشعبية منقطعة النظير في أوساط المحافل البرمجيـة على الرغم من قدمها.

فلسفة تصميم سي بلس بلس:

١. أن تكون لغة متعددة الأغراض، ثابتة في الكتابة، ذات كفاءة مثل سي ومتنقلة مثل سي.

٢. أن تكون ذات نماذج متعددة: إجرائية، كائنية، تجريد البيانات.

٣. أن تكون ذات توافق مع لغة سي.

٤. تجنب الخصائص التي تعتمد على منصة معينة.

٥. مصممة للعمل بدون الحاجة لبيئة برمجة معقدة.

٦. إتاحة الخيارات للمبرمج، حتى لو كانت هناك احتمالية أن يخطئ المبرمج في الاختيار.

لغات أثرت على لغة سي بلس بلس:

- C
- ALGOL68
- Simula
- CLU
- ML
- Ada 83

لغات تأثرت بلغة سي بلس بلس:

- Perl
- Java
- Falcon
- php
- C#
- Ada95
- Lua
- D

خصائص اللغة:

- لغة برمجة متعددة النماذج.
- أسلوب الكتابة ثابت.
- لغة مترجمة.
- لغة وسطية المستوى.
- متوافقة مع لغة سي (ليس ١٠٠٪).
- لا يوجد بها جامع قمامة.

أهم مجالات التطبيق:

- سي بلس بلس لغة متعددة الأغراض، ذات شعبية واسعة، ومن الخيارات المفضلة في المشاريع العملاقة. يوجد لها الكثير من المترجمات. وبرمج بواسطتها الكثير جداً من البرامج مثل: برامج شركة أدوبي من فوتوشوب واكروبات وإليستريتور وإنديزاين، برنامج التصميم الثلاثي الأبعاد العملاق "مايا"، برنامج أوتوكاد، متصفح كروميوم، متصفح الفايرفوكس، عميل البريد (ثاندر برد)، نظام الهواتف سيمبيان، الواجهة الرسومية الأنيقة KDE،إلخ. وقد طُورت العديد والعديد من الألعاب باستخدام هذه اللغة مثل World of Warcraft ، Civilization and the Jews وغيرهم.

مثال برمجي:

تخمين الرقم بين ١ و ١٠.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
int main()
{
srand(time(0));
int n = 1 + (rand() % 10);
int g;
std::cout << "I'm thinking of a number between 1 and 10.\nTry to guess it! ";
while(true)
{
std::cin >> g;
if (g == n)
break;
else
std::cout << "That's not my number.\nTry another guess! ";
}
std::cout << "You've guessed my number! ;
return 0;
}
```

٢٠. Scala

لغة البرمجة سكالا لغة حديثة تجمع بين خصائص وقوة البرمجة الكائنية والبرمجة الوظيفية والاسم اختصار لـ "Scalable Language" والذي يلمح إلى قدرة اللغة على التوسع بحسب احتياجات المستخدم وليس هذا الأمر المستبعد كونها تبني بقوة على اللغة العملاقة جافا رائدة هذا المجال.

بدأ تصميم هذه اللغة في سنة ٢٠٠١ على يد مارتن اودرسكي في معامل EPFL. وقد كان مارتن عمل على Funnel وهي لغة برمجة تجمع بين البرمجة الوظيفية ولغة التمثيل الرياضية Petri Nets، هذا بالإضافة إلى أنه عمل على javac (مترجم جافا) و Generic Java. وقد صدرت اللغة في نهاية ٢٠٠٣ وبداية ٢٠٠٤ على منصة جافا ومن ثم على منصة Net. في شهر جون ٢٠٠٤. وصدرت النسخة الثانية في مارس من سنة ٢٠٠٦.

سكالا تعمل على منصة جافا ومتوافقة مع برامج جافا ولديها القدرة أيضاً على العمل فوق منصة J2ME (منصة جافا للجوالات).

طريقة عمل سكالا تشابه طريقة عمل جافا حيث يقوم مترجم سكالا بتوليد ByteCode (لغة وسطية) يشبه إلى حد كبير ما يولده مترجم جافا. بل، يمكن إرجاع أكواد سكالا (Decompile) إلى أكواد جافا مع بعض الاستثناءات. أما بالنسبة إلى JVM فهي لا تفرق بين أكواد جافا وسكالا، الفرق الوحيد هو في مكتبة إضافية واحدة Scala-library.jar.

لغات أثرت على لغة سكالا:

- Java
- Pizza
- Scheme
- Smalltalk
- Objective Caml
- Standard ML
- Haskell

لغات تأثرت بلغة سكالا:

- Fantom
- Ceylon
- Lasso
- Kotlin

خصائص اللغة:

- أسلوب الكتابة ثابت.
- لغة كائنية صرفة.
- لغة عالية المستوى.
- لغة برمجة متعددة النماذج أمرية (كائنية ووظيفية).
- تتمتع ب ScalaTest بالإضافة إلى دعم JUnit وغير ذلك للاختبارات.
- تتمتع بأهم خصائص اللغات الوظيفية

Closures، Higher-order functions, Anonymouse functions, currying.

أهم مجالات التطبيق:

- إطار Lift وهو إطار برمجة ويب مجاني يشابه في هدفه Ruby on Rails. وبما أن Lift مكتوب بسكالا هذا يعني القدرة على الاستفادة من مكتبات جافا وحاويات الويب الخاصة بها في برامج Lift. في أبريل ٢٠٠٩ أعلنت تويتر أنها قامت بنقل أجزاء كبيرة من روبي إلى سكالا وأنها تعمل على نقل ما تبقى، هذا أثار بعض الانتقادات لروبي أون ريلز وأنها تعاني مشاكل في إدارة المشاريع العملاقة. أيضاً تطبيق الويب WattzOn كُتب بشكل كامل باستخدام سكالا. وموقع Fourthsquare يستخدم سكالا.

الترخيص القانوني:

BSD

موقع اللغة:

http://www.scala-lang.org/

مثال برمجي:

تخمين الرقم بين ١ و ١٠.

```
val n = (math.random * 10 + 1).toInt
print("Guess the number: ")
while(readInt! n) print("Wrong! Guess again: ")
println("Well guessed! )
```

٢١. Self

طُورت لغة Self بواسطة David Ungar و Randall Smith في سنة ١٩٨٦ في معامل Xerox Parc. كان هدفهم الأساسي هو دفع وتطوير فن البرمجة الكائنية وذلك بعد أن نشرت معامل زيروكس لغة Smalltalk-80 وبدأت الشركات بالاهتمام الجدي بها. بعدها انتقل الاثنان إلى جامعة ستانفورد وواصلا العمل على اللغة حيث استطاعا في سنة ١٩٨٧ أن ينشئا أول مترجم للغة.

في سنة ١٩٩٠ صدرت أول نسخة للاستخدام وفي سنة التالية انتقل فريق التطوير إلى شركة صن ميكروسيستمز. وتتابعت الإصدارات إلى أن وقفت في الإصدارة الرابعة في سنة ١٩٩٥. الإصدار ٤.٣ نُشر في سنة ٢٠٠٦ حيث أصبح يعمل على نظام ماك وسولارس. وفي الإصدار الجديد أُضيف دعم لينكس بالإضافة إلى الماك من قبل مجموعة من المطورين الأصليين مع مجموعة متطوعة من المبرمجين.

سيلف لغة برمجة كائنية تعتمد على مبدأ النماذج Prototypes وقد استخدمت في الأغلب كنظام تجريبي لبناء وتصميم لغات البرمجة في الثمانينات والتسعينات. في سنة ٢٠٠٦ استمر تطوير لغة سيلف من خلال مشروع Klein وهي منصة مكتوبة كلياً بلغة سيلف.

العديد من تقنيات الترجمة في الوقت المناسب Just-in-time compilation (أو JIT اختصاراً) طُورت وحُسنت من خلال الأبحاث التي تمت في هذه اللغة لتصل إلى سرعة تقارب نصف سرعة أكواد سي المخصصة. هذه التقنيات بالطبع لاقت رواجاً واسعاً واستخدمت في جافا من خلال Hotspot VM.

لغات أثرت على لغة سيلف:

- SmallTalk

لغات تأثرت بلغة سيلف:

- NewtonScript
- Cel
- Agora
- Lisaac
- Lua
- Factor
- Javascript
- Rebol
- Squeak

خصائص اللغة:

- لغة كائنية التوجه.
- لغة تعتمد النماذج Prototypes.
- توفر دعم Traits.
- لغة عالية المستوى.
- لغة ديناميكية.
- تحتوي على جامع قمامة.
- تحتوي على UI.

الموقع الرسمي:

http://www.selflanguage.org

مثال برمجي:

```
(| parent* = obj1. width = 5. width: = <-.
height = 9. height: = <- |)
(| parent* = obj1. width <- 5. height <- 9 |)
```

الفصل الثالث

أنت تعرف الكثير! اكتب برامجك!

الكثير منا يعرف مبادئ البرمجة والكثير منا قد يكون تخرج من الجامعة بدرجة بكالوريوس في تقنية المعلومات أو حتى دبلوم ولكن في النهاية لا يستطيع أن يبرمج برنامجاً متكاملاً أو لنقل أي مشروع برمجي فنحن على ما نملكه من معرفة ينقصنا معرفة كيفية دمج ما تعلمناه في وحدة واحدة لنخرج بشيء جميل. لنأخذ مثالاً: شخص يعرف أوامر قواعد البيانات ولكنه لا يستطيع أن ينشئ برنامج مدير مقالات، لماذا؟

ماذا نعني ببرنامج متكامل؟

قطعاً لا يقصد بمتكامل بمعنى كامل لا نقص ولا خلل فيه فلا يوجد مثل هذا البرنامج حتى تلك البرامج التي تقوم عليها شركات تضخ الملايين من الدولارات لمبرمجيها، ولكن ما نقصده أنه متكامل بحيث يؤدي مجموعة من الوظائف المختلفة، المبرمجة بمهارات مختلفة، المؤطرة في إطار واحد سهل الاستخدام.

لهذا يمكن أن نرجع سبب عجز الكثير منا عن برمجة برنامج متكامل إلى عدم الإلمام بكل جوانب تطوير البرامج التي لا تشمل البرمجة فحسب بل التخطيط والاختبار والتصحيح والتعامل مع المستخدم، إلخ. فكتابة الأكواد ما هي إلا جزء من عملية البرمجة وهي الجزء الممتع ولكن مع تطوير برنامج متكامل نحتاج إلى أكثر من المتعة نحتاج إلى الروتين والتعامل مع أشياء لا يلاحظها المستخدم النهائي ولكنها تعمل بصمت خلف الستار.

هذه الأشياء المملة تشمل مثلاً رسم الواجهة الرسومية والتعامل مع كل الأحداث Events بشكل مناسب والتعامل مع الأخطاء لا تجاهلها، فمن السهل أن نأخذ قيمة من المستخدم ولكن المشكلة والملل تبدأن حينما نريد أن نتأكد من أن المستخدم أدخل القيمة المناسبة من حيث النوع مثلاً.

إذاً نلاحظ أن برنامج ذو واجهة رسومية بسيط يأخذ عدد من المستخدم ويقسمه على آخر عوضاً من أن يكون مهمة سريعة وسهلة أصبح أمراً مملاً ورتيباً حيث سنكتب الكثير من الأسطر البرمجية لرسم الواجهة والتأكد من ظهورها بشكل صحيح ثم نتأكد من أن المستخدم أدخل رقماً وليس حرفاً ونتأكد أن الرقم المقسوم عليه ليس صفراً، إلخ.

عوداً على بدء، الكثير منا يجهل أن ما يملكه من معلومات بسيطة بنظره هي كافية جداً لبناء برنامج متكامل، المسألة تكمن في معرفة الأوامر والمهارات المطلوبة ومتى استخدمها.

لقد واجهت الكثير من الطلاب يلقون باللوم على المناهج الدراسية ويتهمونها بالنظرية المحضة ولكنهم يغفلون أن ما تعلموه كاف جداً لبدء مشاريعهم الخاصة، هذا لا يعني أن الأمر بسيط جداً ولا يحتاج لأي جهد ولكن بشكل عام المشروع بشكله الكلي يمكن كتابته بهذه الأساسيات وقد نحتاج إلى البحث والسؤال والاستعانة بمصادر خارجية في بعض الأحيان وهذا هو المطلوب حيث سنكتسب مهارات جديدة قد لا نستخدمها في هذا المشروع بالذات ولكن تعود بالنفع في مشاريع أخرى فعندما أبحث عن دالة تقوم بعمل معين من خلال بحثي قد أقرأ عن دالات مشابهة أو حتى مختلفة استرجعها في وقت لاحق إن احتجتها.

وسنحاول في هذا الموضوع السير خطوة خطوة لكتابة برنامج بسيط ولكن سنحاول أن نجعله متكاملاً بحيث نرى الخطوات الأساسية التي نحتاج لها لبناء برنامجنا الشخصي.

١. حدد فكرة البرنامج!

هذه المسألة بديهية وربما تبدو ساذجة ولكن الصعوبة التي نجدها في كتابة مواضيع التعبير والإنشاء تكمن في عدم قدرتنا على تحديد موضوع محدد وشيق وهذه العقبة أي عقبة الخطوة الأولى كبيرة جداً فالشاعر أعطه فقط مطلع القصيدة وسيكملها تلقائياً. بالمثل لا يمكن أن نبرمج بدون أن نعرف ماذا نريد بالضبط وما هو الهدف الذي نرجوه.

من الأشياء التي تجدر الإشارة إليها هنا هي أن الفكرة لا يجب أن تكون فريدة من نوعها ولم يسبق لاحد أن قام بها، المسألة ببساطة ابحث عن فكرة أو مشروع تحس بحاجة شخصية له مثلاً هناك الكثير من برامج مدراء الأخبار المتوافرة ولكن منذ فترة ولدي فكرة عمل برنامج مدير أخبار يكون تركيزه فقط على نشر مقالاتي على شكل مواضيع صالحة للطباعة مباشرة. الكثير من البرامج تبدأ من حاجة المبرمج الشخصية لها ومن ثَم تكبر لشيء هو لم يتوقعه، أو قد لا تكبر! المهم أن البرنامج يسد حاجة شخصية لي، ومنه انطلقت مشاريع كثيرة والذي يعرف بمصطلح "Scratch Your Own Itch".

التطبيق (فكرة البرنامج): في هذا الموضوع مبدئياً قررت أن نبرمج عميل لتويتر على سطح المكتب، ولكني عدلت برأيي واخترت أن يكون قاموساً أو لنقل برنامج ترجمة.

٢. خطط للبرنامج مسبقاً.

فبدلاً من إنشاء جداول قاعدة البيانات مباشرة هكذا ومن ثم إعادة كتابتها كلما أردت إضافة ميزة بسيطة وإعادة كتابة الأكواد البرمجية التابعة لهذه الجداول، خطط مسبقاً للميزات والاختيارات المطلوبة وستقل الحاجة إلى التعديل بعد ذلك.

لا بأس باستخدام برامج تخطيط جداول قواعد البيانات وطريقة اتصالها وأيضاً استخدام UML لتخطيط سير البرنامج أو تخطيط الكلاسات وطريقة تفاعلها. ولكن لا يجب أن نعقد المسألة أكثر من اللازم فإذا كان البرنامج بسيطاً فعلاً فلا داعي لإضاعة الكثير من الوقت في هذه المرحلة.

أيضاً يمكن استخدام التخطيط اليدوي بقلمك ودفتر ملاحظاتك! وهذه طريقتي المفضلة خاصة إذاً كان البرنامج صغيراً أو متوسط الحجم لأنني أجد متعة في ذلك فالمهم أن نجعل من العملية متعة خاصة إذا كان المشروع شخصي ولا ينتظر منه ربح مادي مقابلاً لأتعابك!

التطبيق (التخطيط): طبعاً هناك الكثير من الخيارات تعتمد على نقاط تلي هذه النقطة لذلك لن ندخل في تفاصيل المخطط الآن ولكن بما لدينا من معلومات الآن يمكننا أن نرسم مخططاً بسيطاً لسير البرنامج باستخدام UML أو هكذا على طريقة الخوارزميات:

> حدد اللغات المطلوبة للترجمة ← اطلب النص من المستخدم ← اتصل بقاعدة البيانات ← ارجع قيمة النص المترجم ← أعد العملية.

بسيط أليس كذلك؟ يبدو ذلك في بداية الأمر!

٣. حدد أدواتك واعرف قدراتك.

تحديد الأدوات المناسبة للمهام المناسبة هو مرحلة حساسة في بناء أي مشروع.

- ما هي اللغة البرمجية المناسبة؟
- ما هو الـ IDE المناسب؟
- ما هي المكتبات المساعدة التي سأحتاجها؟
- ما هي المصادر المتوافرة للدعم؟
- ما هي آلية التعامل مع المستخدم؟
- ما هي قاعدة البيانات المستخدمة؟

وغيرهم من الأسئلة، هذه الأسئلة حساسة جداً وتضمن إلى حد كبير أن لا تقع في متاعب كثيرة في مرحلة التطبيق، وتختلف الإجابات بالطبع حسب متطلبات المشروع البرمجي.

السؤال الأول مثلاً في حال كان المشروع تطبيق ويب هل C خيار مناسب؟ في أغلب الأحيان لا، في حال برمجة برنامج ويدجت هل هاسكل مناسبة؟ لا أعتقد ذلك، PHP وبيرل وروبي خيارات أفضل في الحالة الأولى ولغة JavaScript في الحالة الثانية.

السؤال الثالث ما هي المكتبات التي سأحتاجها؟ كلما كان البرنامج أكبر ويحتاج إلى أشياء معقدة كلما كانت الحاجة إلى المكتبات أكثر، قبل أن ابدأ في مشروع كتابة برنامج سطح مكتب يجب أن أدرس الخيارات التي تقدمها لي لغة البرمجة هل توفر لي GTK، Wxwidgets، إلخ؟ أي مكتبة سأختار هل سيكون برنامجي لويندوز أو لينكس أو ماك؟

ما هي المصادر المتوافرة؟ هل إذاً واجهتني مشكلة سأجد من يساعدني؟ هل تتوافر مصادر غنية؟ هل يوجد توثيق كامل؟

لا تنسى أن تعرف قدراتك، بعض المكتبات ليست مجرد مكتبات بل عالم متشعب في حد ذاتها فتوافر مكتبة SDL لا يعني أنني أستطيع كتابة لعبة فقط من خلال قراءة التوثيق بل تتعدى إلى مهارات وتقنيات ومفاهيم إضافية تحتاج إلى وقت كبير لتعلمها يوازي ربما الوقت الذي قضيته لتعلم لغة البرمجة نفسها!

التطبيق (حدد أدواتك): سأكتفي بالإجابة عن الأسئلة المطروحة سلفاً فقط، وذلك على حسب حاجة برنامجنا.

ما هي اللغة البرمجية المناسبة؟

بما إنني أجيد بيرل والمشروع شخصي واللغة مناسبة لهذا نوع من المهام فلا بأس من استخدامها.

ما هو الـ IDE المناسب؟

سأختار Padre بالإضافة إلى wxGlade وذلك لأن Glade سيرسم لنا الواجهة الرسومية و Padre سيساعدنا في تطوير أكواد بيرل خاصة أنه برنامج مكتوب بها وبمكتبة wx!

ما هي المكتبات المساعدة التي سأحتاجها؟

سنحتاج بعض المكتبات ولكن المهم الآن هو مكتبة wxPerl التي ستوفر لنا إمكانية إنشاء الواجهة الرسومية، طيب لماذا لا نستخدم GTK أو QT أو حتى Swing؟

هنا نحتاج أن نعمل دراسة سريعة قبل اتخاذ القرار فمكتبة كيوت مكتبة ممتازة جداً ولكن في بيرل هي خيار سيئ لقلة المصادر وقدم الإصدار المتوافر. GTK أيضاً ممتازة لو كنا سنكتفي بنظام لينُكس ولكن ربما أحب أن انقل البرنامج إلى ويندوز وتشغيل هذه المكتبة هناك ليس بالأمر السهل. سوينج؟ سنحتاج إلى مكتبات إضافية كثيرة لكي نستطيع تشغيلها من بيرل فلا داعي لكل هذا، Tk أفضل المكتبات المتوفرة لبيرل من ناحية التوثيق ولكنها لا تدعم العربية! إذاً الخيار الأفضل هو wxPerl فالتوثيق موجود والمكتبة قوية جداً ومتعددة المنصات في حال رغبنا في تشغيلها في أي نظام تشغيل لكي تظهر بنفس شكل برامج نظام التشغيل.

ما هي المصادر المتوافرة للدعم؟

بيرل لا يوجد لها دعم مادي خاصة في عالمنا العربي لذلك قد لا تكون خياراً مناسباً للشركات هنا ولكن المصادر المتوافرة ممتازة جداً للمشاريع الشخصية من كتب وتوثيق ومنتديات وغرف مساعدة ومجموعات بريدية.

ما هي آلية التعامل مع المستخدم؟

سطر الأوامر؟ صفحات ويب؟ واجهة رسومية؟ يبدو أنكم خمنتم الخيار الثالث لأننا تكلمنا عن المكتبة التي سنستخدمها!

ما هي قاعدة البيانات المستخدمة؟

ملف عادي Flat File؟ ماي سيكوال؟ أوراكل؟ SQLite؟ يمكن اختيار الفلات فايل في المشاريع الصغيرة والمتوسطة ولكنها خيار سيئ للمشاريع الكبيرة ولن اختارها لكي لا أحتاج للكثير من الأسطر البرمجية لفتح وقفل وإغلاق الملفات النصية.

مايسكوال لن استخدمها لان برنامجي لا يحتاج إلى خادم فهو ليس تطبيق ويب. أوراكل ليس لدي المال الكافي! قاعدة بيانات SQLite كنت سأستخدمها لو كنت سأصنع قاعدة الكلمات بنفسي ولكن لأني كسول سأستخدم قاعدة كلمات جاهزة!

إذن الآن لدي خيارين أن استخدم قاعدة بيانات جاهزة للكلمات مثل ما توفره Arabeyes وهي من نوعية الملفات النصية أو أن استخدم قاعدة بيانات موجودة على الويب مثل ما توفره BabelFish؟ ولكنه لا يوفر اللغة العربية؛ لذا سأختار مترجم جوجل!

٤. رسم الواجهة الرسومية.

طبعاً يمكن البدء بعملية كتابة الأكواد والدوال الحقيقية أولاً ثم رسم الواجهة الرسومية أو ما تعرف بالانجليزية اختصاراً GUI خاصة لمن يحبون طريقة الاختبار قبل البرمجة وهذا ما فعلته أنا عندما كنت أكتب الأكواد التي سنستخدمها في هذا الموضوع ولكن هنا فضلت أن نرسم الواجهة الرسومية بشكل سريع وننتهي منها أولاً.

التطبيق (رسم الواجهة يدوياً): أولاً وقبل كل شيء ارسم الواجهة بيدك كما تحب أن تظهر لكي ترتاح في عملية وضع الأشياء في أماكنها الصحيحة لاحقاً!

هنا رسمي المتواضع:

ترجم | اللغة الثانية | تبديل | اللغة الأولى

النص المراد ترجمته

النص المُترجم

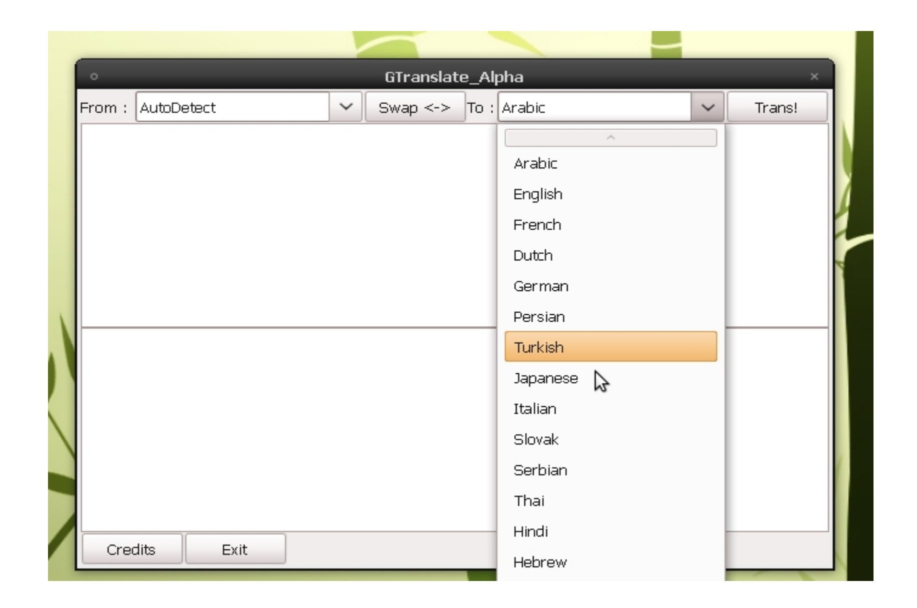
خروج | عن البرنامج

بعدها يمكننا البدء في استخدام wxGlade للرسم الحقيقي ولن أتكلم عن هذه النقطة حيث يوجد العديد من المواد التي تشرح هذه الجزئية وقد قمت بعمل بعضها.

٥. ربط الواجهة الرسومية مع الأكواد الحقيقية.

بعد رسم الواجهة الرسومية باستخدام برنامج wxGlade أو أي برنامج تصميم واجهات مثل Glade و Tk builder، إلخ نأتي إلى مرحلة الربط بين الواجهة والكود البرمجي، قبل أن استرسل يجب أن أنوه إنني كنت من المعارضين لاستخدام برامج رسم الواجهات الرسومية لأنها تضيف أكواد إضافية كثيرة يمكن اختصارها ولكن الميزة التي تجعلني أفضل استخدامها هو أولاً سرعة التصميم بالطبع، فننتهي من هذه المرحلة المملة بسرعة وأيضاً توفيرها آليات جيدة لفصل الكود البرمجي عن الكود المرئي وذلك سواء باستخدام ملف وصفي منفصل (مثلاًXML) أو حتى تضمين الكود البرمجي في الكود المرئي ولكن بتوفير آليات مريحة لتحديث الكود المرئي بشكل منفصل عن الكود البرمجي.

نعود مرة أخرى، انتهينا من تصميم الواجهة الرسومية وهكذا تبدو:



ماذا نفعل الآن؟

الخطوة الأساسية الآن هي أن نربط الواجهة بالكود وذلك يتم عن طريق وضع متنصت Action listener (Handler) لكل حدث Event.

عندما أشاهد كل جزء من برنامجي أعرف أنه يمكن أن تحدث أحداث كثيرة في برنامجي ولكن يمكنني فقط الاهتمام بالأحداث التي لها تأثير مباشر على سير البرنامج، مثلاً يمكنني وضع متنصت لتحرك مؤشر الفأرة على منطقة الكتابة وعليه أخذ تصرف معين ولكن لا يوجد كثير جدوى من فعل ذلك في برنامجنا الآن.

إذاً سألخص أهم النقاط التي يجب أن اهتم بها كالتالي:

١. سأضع متنصت للنقر على زر الخروج لإنهاء كل عمليات البرنامج.

٢. سأضع متنصت للنقر على زر Credit كي أظهر معلومات البرنامج.

٣. سأضع متنصت لزر Swap ولكي يقوم باستبدال أماكن اللغتين.

٤. سأضع متنصت لزر Trans ولكي يقوم بمهمة الترجمة وعرضها في مكانها الصحيح.

ملاحظات:

١. الجزء الذي سيظهر فيه النص المترجم سيُقفل بحيث لا يمكن للمستخدم الكتابة فيه ولكن يمكنه النسخ منه وأيضاً سيتم إظهار رسائل الأخطاء في هذا المكان.

٢. لدينا خيارات كثيرة في كيفية توفير آلية اختيار اللغات ولكن سنختار ComboBox لسهولته ولتوفيره المكان فهو لا يأخذ مكانا كبيراً في البرنامج وإنما يتوسع فقط في حالة النقر عليه ليظهر اللغات المتوفرة (سنقوم بإضافة أكثر من ٣٠ لغة ولكن لن نضيف كل اللغات التي يوفرها جوجل فبعضها إلى الآن مازال تجريبياً).

٣. لن نضع متنصتاً خاصاً بكل صندوق اختيار بل سنطلبها يدوياً في دالة الترجمة، لأن هذا الحدث ليس مهماً إلا في حالة النقر على زر الترجمة.

التطبيق: سيكون لدينا ملف بيرل جاهز بعد توليده من برنامج wxGlade حيث سنكون جاهزين لبدء الربط بين البرنامج والواجهة من خلال تطبيق Implement الدوال التي قمنا بالإعلان عنها مبدئياً:

```
#! usr/bin/perl -w --
# generated by wxGlade 0.6.3 on Fri Oct 8 21:50:29 2010
# To get wxPerl visit http://wxPerl.sourceforge.net/
use Wx 0.15 qw[:allclasses];
use strict;
package MyFrame1;
use Wx qw[:everything];
use base qw(Wx::Frame);
use strict;

sub new {
my($self, $parent, $id, $title, $pos, $size, $style, $name) =
@_;
$parent = undef unless defined $parent;
$id = -1 unless defined $id;
$title = "" unless defined $title;
$pos = wxDefaultPosition unless defined $pos;
$size = wxDefaultSize unless defined $size;
$name = "" unless defined $name;
# begin wxGlade: MyFrame1::new
$style = wxCAPTION|wxCLOSE_BOX|wxFRAME_NO_TASKBAR|
wxCLIP_CHILDREN
unless defined $style;
$self = $self->SUPER::new($parent, $id, $title, $pos, $size,
$style, $name);
$self->{label_1} = Wx::StaticText->new($self, -1, "From: ",
wxDefaultPosition, wxDefaultSize,);
$self->{origin} = Wx::ComboBox->new($self, -1, "",
wxDefaultPosition, wxDefaultSize, ["AutoDetect", "Arabic",
"English", "French", "Dutch", "German", "Persian", "Turkish",
"Japanese", "Italian", "Slovak", "Serbian", "Thai", "Hindi",
"Hebrew", "Spanish", "Greek", "Russian", "Swedish", "Croation",
"Polish", "Portuguese", "Filipino", "Irish", "Malay",
"Belarusian", "Czech", "Norwegian", "Swahili", "Catalan",
"Bulgarian", "Korean", "Indonesian", "Chinese", "Vietnamese"،
"Yiddish"، "Afrikaans"], wxCB_DROPDOWN|wxCB_SIMPLE|
```

```
wxCB_DROPDOWN|wxCB_READONLY|wxCB_SORT);
$self->{Swap} = Wx::Button->new($self, -1, "Swap <->");
$self->{label_2} = Wx::StaticText->new($self, -1, "To:",
wxDefaultPosition, wxDefaultSize,);
$self->{destination} = Wx::ComboBox->new($self, -1, "",
wxDefaultPosition, wxDefaultSize, ["Arabic", "English",
"French", "Dutch", "German", "Persian", "Turkish", "Japanese",
"Italian", "Slovak", "Serbian", "Thai", "Hindi", "Hebrew",
"Spanish", "Greek", "Russian", "Swedish", "Croation", "Polish",
"Portuguese", "Filipino", "Irish", "Malay", "Belarusian",
"Czech", "Norwegian", "Swahili", "Catalan", "Bulgarian",
"Korean", "Indonesian", "Chinese", "Vietnamese", "Yiddish"،
"Afrikaans"، "Ukrainian"], wxCB_DROPDOWN|wxCB_SIMPLE|
wxCB_DROPDOWN|wxCB_READONLY|wxCB_SORT);
$self->{Trans} = Wx::Button->new($self, -1, "Trans! );
$self->{origin_text} = Wx::TextCtrl->new($self, -1, "",
wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE|wxHSCROLL);
$self->{translated_text} = Wx::TextCtrl->new($self, -1, "",
wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE|wxTE_READONLY);
$self->{Credits} = Wx::Button->new($self, -1, "Credits");
$self->{Exit} = Wx::Button->new($self, -1, "Exit");
$self->__set_properties();
$self->__do_layout();
Wx::Event::EVT_COMBOBOX($self, $self->{origin}->GetId,
\&OnSelection);
Wx::Event::EVT_BUTTON($self, $self->{Swap}->GetId, \&OnSwap);
Wx::Event::EVT_COMBOBOX($self, $self->{destination}->GetId,
\&OnSelection);
Wx::Event::EVT_BUTTON($self, $self->{Trans}->GetId, \&OnTrans);
Wx::Event::EVT_BUTTON($self, $self->{Credits}->GetId,
\&OnCredit);
Wx::Event::EVT_BUTTON($self, $self->{Exit}->GetId, \&OnExit);
# end wxGlade
return $self;
}
sub __set_properties {
```

```
my $self = shift;
# begin wxGlade: MyFrame1::__set_properties
$self->SetTitle("GTranslate");
$self->{origin}->SetSelection(0);
$self->{destination}->SetSelection(0);
$self->{origin_text}->SetMinSize(Wx::Size->new(428, 77));
# end wxGlade
}

sub __do_layout {
my $self = shift;
# begin wxGlade: MyFrame1::__do_layout
$self->{sizer_2} = Wx::BoxSizer->new(wxVERTICAL);
$self->{sizer_3} = Wx::BoxSizer->new(wxVERTICAL);
$self->{sizer_4} = Wx::BoxSizer->new(wxHORIZONTAL);
$self->{sizer_5} = Wx::BoxSizer->new(wxHORIZONTAL);
$self->{sizer_5}->Add($self->{label_1}، 0,
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_5}->Add($self->{origin}، 0,
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_5}->Add($self->{Swap}، 0,
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_5}->Add($self->{label_2}، 0,
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_5}->Add($self->{destination}، 0,
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_5}->Add($self->{Trans}، 0,
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_3}->Add($self->{sizer_5}، 1, wxEXPAND, 0);
$self->{sizer_3}->Add($self->{origin_text}، 0, wxEXPAND|
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_3}->Add($self->{translated_text}، 0, wxEXPAND|
wxALIGN_CENTER_HORIZONTAL|wxALIGN_CENTER_VERTICAL, 0);
$self->{sizer_4}->Add($self->{Credits}، 0, wxALIGN_RIGHT, 0);
```

```
$self->{sizer_4}->Add($self->{Exit}، 0, wxALIGN_RIGHT, 0);
$self->{sizer_3}->Add($self->{sizer_4}، 1, wxEXPAND, 0);
$self->{sizer_2}->Add($self->{sizer_3}، 1, wxEXPAND, 0);
$self->SetSizer($self->{sizer_2});
$self->{sizer_2}->Fit($self);
$self->{sizer_2}->SetSizeHints($self);
$self->Layout();
# end wxGlade
}

sub OnSelection {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnSelection <event_handler>
warn "Event handler (OnSelection) not implemented";
$event->Skip;
# end wxGlade
}

sub OnSwap {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnSwap <event_handler>
warn "Event handler (OnSwap) not implemented";
$event->Skip;
# end wxGlade
}

sub OnTrans {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnTrans <event_handler>
warn "Event handler (OnTrans) not implemented";
$event->Skip;
# end wxGlade
}
```

```
sub OnCredit {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnCredit <event_handler>
warn "Event handler (OnCredit) not implemented";
$event->Skip;
# end wxGlade
}

sub OnExit {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnExit <event_handler>
warn "Event handler (OnExit) not implemented";
$event->Skip;
# end wxGlade
}

# end of class MyFrame1
1;
1;
package main;
unless(caller){
local *Wx::App::OnInit = sub{1};
my $app = Wx::App->new();
Wx::InitAllImageHandlers();
my $frame_2 = MyFrame1->new();
$app->SetTopWindow($frame_2);
$frame_2->Show(1);
$app->MainLoop();
}
```

قد يسألني البعض الآن كنت تدعي أن العملية ستكون سهلة وأننا بمعرفة الأساسيات يمكن أن ننشئ برنامج فما هذه الأكواد الطويلة والمعقدة؟

ولكن في حقيقة الأمر إن استخدامنا لبرامج تصميم الواجهات وفر لنا بعض المزايا:

١. إزالة الحاجة إلى كتابة الأكواد الطويلة للوصول إلى شكل مبدئي للبرنامج! خاصة مثلاً لو كانت المكتبة سوينج مثلاً.

٢. تسهيل عملية إعادة ترتيب مواضع الأشياء ورسمها.

٣. إزالة الحاجة إلى حفظ خصائص العناصر المرئية والأحداث المرتبطة بها. فأنا لست مضطراً لأن أحفظ ما هي خصائص العنصر (زر) وما هي نوعية الأحداث التي يستجيب لها.

٤. سهولة تعلمها فهي لا تأخذ إلا قليلاً جداً من الوقت حتى يتم تعلمها بخلاف تعلم المكتبة نفسها. بالإضافة إلى كون تعلم برنامج تصميم واحد يسهل عملية الانتقال إلى برنامج تصميم آخر إلى حد كبير جداً.

٥. وأخيراً الشيء المهم وهو إزالة عناء تعلم المكتبة الرسومية خاصة فيما يتعلق برسم العناصر وطريقة العرض، مثلاً هذا الكود الآن ليس كل مبرمج بيرل يعرف كتابته بنفسه ولكن الآن بسهولة يستطيع أن يعمل تطبيق لكل دالة فقط وينتهي من البرنامج.

إذاً إلى الآن نحن في الحقيقة لم نبرمج بل نصمم كأي برنامج تصميم مثل GIMP أو فوتوشوب أو فرونت بيج، إلخ. وأغلب المكتبات الرسومية الآن يتوافر لها عدة برامج تصميم تزيل هذا العبء عن المبرمج.

٦. كتابة الكود الحقيقي.

مع هذه المرحلة يمكننا أن نقول بأن البرمجة الحقيقية بدأت فمثلاً لو أن برنامجنا يقوم بعمليات حسابية معقدة فكل ما فعلناه إلى الآن مجرد تمهيد والبداية الفعلية هي في كتابة هذه المعادلات الرياضية في الدوال الموجودة في البرنامج.

الآن، بما أن البرنامج ذو واجهة رسومية فهو يحتوي على دوال لا بأس بها افتراضياً وذلك للتعامل مع كل حدث مهم في البرنامج. وبما أن البرمجة الحدثية لا تبدأ من اتجاه معين ولا تتبع سير عمل ثابت (مثلاً يمكن أن نبدأ برنامج رسومي ما باختيار فتح ملف جديد أو يمكن أن نبدأ باستحضار ملف تم إنشاؤه سابقاً ومن ثم تتوالى الأحداث بشكل مختلف في كل مرة تشغيل تقريباً).

فأنا شخصياً أفضل البدء في تطبيق الدوال التي:

- لا تحتاج إلى وقت وجهد طويل لإنشائها.
- لا تؤثر أو تتفاعل مباشرة مع الأحداث الأخرى في البرنامج (أو على الأقل تتفاعل وتتأثر بشكل أقل من غيرها).

وبالنظر إلى برنامجنا يمكن ترتيب الدوال (من الأقل إلى الأكثر) هكذا:

١. onExit
٢. onCredit
٣. onSwap
٤. onTrans

النقطة التي احب أن أشير إليها هنا هو أن الأكواد التي سأكتبها في التطبيق ليست بهدف تعليم لغة البرمجة بيرل أو مكتبة wx وإنما هي وسيلة لتجسيد المبادئ المجردة والأفكار فقط وبالتالي لا تهتم بكيفية كتابة الأمر المعين ولكن ركز على خطوات الوصول إلى نتيجة معينة.

التطبيق (كتابة الأكواد):

أولاً: نبدأ بدالة الخروج وهي دالة بسيطة مهمتها إنهاء جميع عمليات البرنامج والخروج بشكل سليم حسب طلب المستخدم:

```
sub OnExit {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnExit <event_handler>
$self->Close();
# end wxGlade
}
```

ثانياً: onCredit ومهمتها هي توفير معلومات عن البرنامج مثل تاريخ الإصدارة ورقمها وحقوق النشر مثلاً إلى آخره) ويمكن أن تكتب بطرق مختلفة كثيرة وفي مثالنا سأقوم بإنشاء مربع حوار Dialog box يقوم بمهمة إظهار هذه المعلومات بحيث عندما ينقر المستخدم على زر "Credit" يقفز له مربع حوار صغير فيه معلومات بدائية عن البرنامج، وعندما ينتهي المستخدم من القراءة ينقر زر موافق ليرجع إلى البرنامج الأساسي.

هذا كل ما في الأمر:

```
sub OnCredit {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnCredit <event_handler>
my $credit = Wx::MessageDialog->new(
$self,
"All Credit goes to: \n
Google: http://translate.google.com \n
CPAN: Lingua::Translate - Lingua::Translate::Google \n
wxPerl & wxGlade: perl_sourcer\@yahoo.com \n
Still under testing:)
",
"Credits");
$credit->ShowModal;
# end wxGlade
}
```

ثالثاً: OnSwap وهذه الدالة مهمتها فقط أن تقوم بتغيير أماكن لغتي الترجمة على سبيل المثال إذا كانت اللغة المترجم منها العربية واللغة المستهدفة الانجليزية سيقوم أوتوماتيكياً بتغيير أماكن اللغتين. إذاً بكل بساطة ما نحتاجه هو:

١. نحصل على قيمة اللغة الأولى ونسندها إلى متغير.

٢. نحصل على قيمة اللغة الثانية ونسندها إلى متغير الثاني.

٣. نقوم باستبدال القيم في المتغيرات.

٤. نحدث اللغتين في الواجهة الرسومية بالقيم الجديدة.

```
sub OnSwap {
my ($self, $event) = @_;
# wxGlade: MyFrame1::OnSwap <event_handler>
my $first = $self->{origin}->GetValue();
my $second = $self->{destination}->GetValue();
($first, $second) = ($second, $first);
$self->{origin}->SetValue($first);
$self->{destination}->SetValue($second);
# end wxGlade
}
```

رابعاً: onTrans وهي الدالة التي تقوم بعمل الشيء المهم ألا وهو الترجمة. خطوات العمل كالتالي، في بداية الملف سنستورد المكتبة اللازمة للترجمة:

```
use Lingua::Translate;
```

وأيضاً في بداية الملف وليس داخل هذه الدالة سننشئ كائن جديد من هذه المكتبة بهذا الشكل:

```
Lingua::Translate::config
(
back_end => 'Google'،
referer => 'http://dheeb.wordpress.com'،
format => 'text'،
userip => '192.168.1.1'،
);
```

نعود إلى داخل الدالة، بما أن جوجل لا تتعامل مع اللغات بأسمائها بل باختصاراتها فعلينا أولاً استبدال كل لغة باختصارها، مثلاً Arabic ستصبح ar.

وعليه، أول سطرين:

```
my $tag_1 = getTag($self->{origin}->GetValue());
my $tag_2 = getTag($self->{destination}->GetValue());
```

نلاحظ أننا استدعينا دالة اسمها getTag وهي دالة مساعدة كتبتها لتقوم بعملية الحصول على الاختصار سأضع الكود الخاص بها بعد الانتهاء من هذه الدالة.

الآن ننشئ الكائن بما أن لدينا كل المعلومات:

```
$object = Lingua::Translate->new(src => $tag_1, dest => $tag_2);
```

الآن انتهينا سنأخذ قيمة النص الموجود ومن ثم ننفذ ميثود الترجمة وسنحدث خانة النص المترجم بالقيمة الجديدة:

```
my ($message, $trans);
$message = $self->{origin_text}->GetValue();
$trans = $object->translate($message);
$self->{translated_text}->SetValue($trans);
```

ملحوظة: الدالة المساعدة getTag: كما قلنا هذه الدالة تقوم بمقارنة كل اسم مع اختصاره وترجع قيمة الاختصار. إذاً سنمرر لها قيمة اللغة الموجودة وعليها أن ترجع لنا الاختصار المناسب ويوجد العديد من الأساليب للوصول إلى هذه النتيجة.

وهكذا كتبتها أنا:

```
sub getTag {
my $value = shift;
my %tags = qw/ AutoDetect auto Arabic ar English en French fr
Dutch nl German de
Persian fa Turkish tr Japanese ja Italian it Slovak sk Serbian
sr Thai th
Hindi hi Hebrew iw Spanish es Greek el Russian ru Swedish sv
Croation hr
Polish pl Portuguese pt Filipino tl Irish ga Malay ms Belarusian
be Czech cs
Norwegian no Swahili sw Catalan ca Bulgarian bg Korean ko
Indonesian id
Chinese zh-CN Vietnamese vi Yiddish yi Afrikaans af Ukrainian
uk/;
return my $tag = $tags{$value};
}
```

الآن بحمد الله انتهينا من البرنامج فهو يقوم بوظائفه الأساسية المطلوبة ولكن هل فعلاً البرنامج جاهز للاستخدام؟ بالطبع لا فنحن بعيدون كل البعد أن يكون البرنامج الآن شبه متكامل! لماذا؟ أليس مهمته فقط أن يقوم بالترجمة وانتهينا؟ أليست كل الأزرار تعمل بالشكل المطلوب؟

مع الأسف هذا البرنامج إلى الآن وهو برنامج ناقص بقوة لأنه لم يخضع إلى المرحلة الأكثر أهمية والتي ربما كانت الهدف الأساسي من هذا الموضوع: مرحلة الاختبار والتجريب والتعامل مع الأخطاء والاستثناءات.

إليك هذا المثال، هناك برنامج يفترض أن الإنترنت متوافرة للاتصال بجوجل وإرجاع القيمة دائماً وهذا خطأ جسيم جداً! لا مكان للافتراضات ومع الأسف كثير منا يترك هذه المرحلة إما لأنه غير مبالي أو لأنه لا يعرف أو لأنه لا يدرك أهميتها. ففي مثال الإنترنت هذا، ما سيحدث هو أن البرنامج عندما نطلب منه الترجمة سيتصل بالإنترنت لثواني معدودة فإذا كان هناك اتصال جرت الأمور على خير ما يرام ولكن لو كان المستخدم غير متصل بالشبكة فسينهار البرنامج ويختفي هكذا بدون سابق إنذار بمجرد النقر على زر الترجمة. ربما البعض لا يهمه فهو يفترض أن المستخدم مدرك لوجوب وجود اتصال قبل تشغيل البرنامج ولكن مع الأسف هذه نقطة تحسب على المبرمج وتجعل برنامجه ناقصاً وغير احترافياً، إذن دع عنك الافتراضات وتعامل مع الأخطاء.

٧. التعامل مع الأخطاء والاستثناءات.

قلنا أن من الأهمية بمكان أن يقوم البرنامج بالتعامل مع الأخطاء والاستثناءات التي قد تحصل وقت تشغيل البرنامج، وهذا الأمر هو من الأمور التي تمايز بين المبرمجين فكلما كان المبرمج مهتما بالتفاصيل ومعالجتها كان ذلك أفضل لبرنامجه ولسمعته كمبرمج.

الآن أين نبحث عن الأخطاء والاستثناءات التي يمكن أن تحصل؟ في الغالب هناك حالتين يكثر جداً ظهور الأخطاء والاستثناءات فيها:

- التعامل مع النظام.
- التعامل مع المستخدم.

الحالة الأولى، مثالها عندما نحاول أن نفتح ملف قد تنجح العملية أو قد لا تنجح بسبب وجود قفل أو صلاحية معينة على الملف لذلك يجب أخذ هذا الأمر بالحسبان والتعامل معه.

الحالة الثانية، مثالها طلب قيمة من المستخدم قد نكون نريد قيمة رقمية لعمل حسابات مثلاً فيدخل المستخدم قيمة نصية، عليه يجب أن نتأكد من القيمة المدخلة قبل الشروع في العمل عليها.

في النهاية هناك أنواع من الأخطاء لا يمكن التعامل معها مثل نفاذ الذاكرة من نظام التشغيل فلن يكون هناك مفسر لغة أصلاً كي يتعامل مع هذا الخطأ.

أيضاً في بعض الأحيان لا يتوقع أن نحصي جميع الأخطاء التي قد تحصل خاصة في المشاريع الكبيرة ولكن مهمتنا أن نقللها إلى حد معقول جداً ولا نترك عملية اكتشاف الأخطاء للمستخدم خاصة في برامج الويب التي قد تكون النتيجة غالية جداً حينها إن استخدمت أحد هذه الأخطاء كثغرة في البرنامج. وبما أن لكل لغة برمجة طريقة في التعامل مع الاستثناءات والأخطاء ليس الهدف من التطبيق هنا هو كيفية كتابة الأكواد وإنما البحث عنها وطريقة محاصرتها ومدى أهميتها.

التطبيق (البحث عن الاستثناءات): الآن نأتي إلى برنامجنا ونأخذ كل دالة على حدة ثم نبحث عن وجود الأخطاء والاستثناءات التي يجب التعامل معها.

دالة الخروج لا تحتاج إلى إضافة أكواد ودالة إظهار المعلومات أيضاً، لذلك سأكتفي بالتجريب المباشر وملاحظة سلوك الدالتين عند النقر على الزر المعين. ويبدو أن الاثنتين تعملان بشكل جيد. والمهم أيضاً انهما لا يطلبان أي قيم من المستخدم.

جميل، نأتي إلى دالة onSwap: هكذا مبدئياً من مجرد الملاحظة نعرف أنها قد تسبب مشكلة أساسية جداً تسبب انهيار البرنامج، المشكلة هي في حالة أن قيم مربع الاختيارات الأول لا يساوي قيم مربع الاختيارات الثاني فهناك قيمة موجودة زائدة في اختيارات لغة المصدر وهي AutoDetect هذه القيمة تخبر جوجل أن عليه هو البحث عن اللغة في المصدر وهذه ميزة جيدة فربما نحن لا نعرف اللغة التي نريد الترجمة منها.

ولكن دالة سواب ستقوم بنقل أي قيمة إلى الخانة الأخرى لان القيمة الموجودة في داخل المربع لا تعنيها فهي غير مهتمة بالترجمة وإنما ما يهمها هو فقط تبديل أماكن القيمتين. ولأن خاصية AutoDetect لا تنفع أن تكون قيمة للغة المترجم إليها (فكيف يخمن جوجل ما هي اللغة التي نريد أن نترجم إليها؟) سيظهر خطأ في البرنامج.

ولكي نتعامل مع هذا الخطأ فقط نتأكد من القيمة أنها ليست AutoDetect قبل القيام بعملية التحويل وإذا كانت القيمة فعلاً AutoDetect فسنقوم بإظهار رسالة تنبيه للمستخدم في خانة النص المترجم ننبهه إلى هذا الخطأ ونرجو منه محاولة الترجمة مرة أخرى بعد تغير قيمة AutoDetect وذلك بدلاً من أن ينهار البرنامج بخطأ لا يعلم عنه المستخدم، إذاً التالي:

```
my $first = $self->{origin}->GetValue();
my $second = $self->{destination}->GetValue();
if($first eq "AutoDetect"){
$self->{translated_text}->SetValue("Can't use AutoDetect on destination! Please choose a language & try again…");}
else {
($first, $second) = ($second, $first);
$self->{origin}->SetValue($first);
$self->{destination}->SetValue($second); }
```

دالة getTag أيضاً قد تسبب مشاكلات فهي تستقبل قيمة وترجع قيمة مقابلة لها ولكن لو كانت القيمة المستقبلة لا يوجد لها مقابل؟ ماذا سنفعل الآن؟!

يجب علينا إذاً أن نحرص على إيجاد آلية لضمان عدم إرسال أي قيمة غير القيم التي يوجد لها مقابل، وذلك قد فعلناه سابقاً فعلاً! نعم فعلناه في مرحلة تصميم البرنامج حيث عندما قمنا بتصميم مربعي الاختيارات قمنا بتوفير قيم افتراضية للغات معينة هي فقط ما يوفره البرنامج للمستخدم ولكي لا يقوم المستخدم بإضافة أي لغة غير موجودة أو أي نص لا معنى له قمنا بقفل مربع الاختيارات حيث لا يمكن البتة أن يكتب المستخدم قيمة مربع الاختيار بنفسه وإنما عليه أن يختار من الموجود فقط وفقط، القيمة التي وفرت لنا هذه الميزة في الكود كانت:

```
wxCB_READONLY
```

متابعة رحلة البحث عن الأخطاء!

دالة OnTrans، هذه الدالة هي أهم دوال البرنامج وهي أكثر الدوال عرضةً للأخطاء، النقطة الأولى التي يجب الانتباه إليها أننا بالنقر على زر الترجمة نقوم باستدعاء الدالة وتنفيذ كل خطوات الترجمة حتى لو كان المستخدم لم يدخل نصا للترجمة أصلاً (ترك الخانة فارغة) هذه العملية لا تسبب مشاكل مهمة ولا ترجع أخطاء لأن محرك جوجل سيرجع قيمة فارغة أيضاً وبذلك لن يلاحظ المستخدم أي شيء! ولكن لا معنى لهذا فلماذا نقوم بتنفيذ أوامر لا تعطينا نتيجة في نهاية المطاف؟

لذلك سنعدل على الدالة بحيث أنها لا تقوم بالتنفيذ إلا إذاً كانت هناك قيمة للترجمة وذلك ببساطة يتم بالتأكد من أن قيمة خانة النص المصدر صحيحة True.

إذاً:

```
if ($self->{origin_text}->GetValue())
{
Rest of code here
}
else {$event->Skip;}
```

الآن بكل بساطة سيتجاهل أمر الترجمة إذاً لم يتوافر نص للترجمة.

النقطة الثانية الأكثر خطورة التي تكلمنا عنها سابقاً هي في حال عدم توافر اتصال بالإنترنت سينهار البرنامج وقت النقر على زر الترجمة. قد يسأل البعض لماذا ينهار البرنامج كاملاً؟

ينهار البرنامج لأن دالة الترجمة تحاول إنشاء كائن من نوع جوجل ترانسليتور وهذا يحتاج إلى اتصال بالإنترنت فعندما يفشل البرنامج في إنشاء هذا الكائن سيرجع خطأ من النوع الذي يجب التعامل معه Fatal وليس اختيارياً. طيب لماذا هذا الخطأ يجب التعامل معه؟ لأنه بكل بساطة كل الأسطر البرمجية التي تعتمد على هذا الكائن ستفشل لأنه لا يوجد كائن أساساً.

مثلاً هذا الأمر:

```
$trans = $object->translate($message);
```

نحن نريد تنفيذ ميثود translate الخاص بالكائن ولكن كائن object أساساً غير معرف عندنا لأننا لم ننشئه بنجاح في العملية السابقة.

كيف إذن نتأكد من وجود اتصال بالإنترنت؟ اقترح احد الأعضاء في منتدى بيرل أن نقوم بعملية اختبار للاتصال قبل محاولة إنشاء الكائن.

ولكن هذه العملية مكلفة لأن المستخدم عندما ينقر على زر الترجمة ما سيحدث أن الدالة في بداية الأمر ستختبر وجود الاتصال وهذا يستغرق بضع ثوان ومن ثم تقوم بإنشاء الكائن باتصال جديد وأخيراً تقوم بتنفيذ الميثود وستزيد ثوان الانتظار هكذا.

فكرتي كانت مغايرة، البرنامج كله لا يحتاج اتصال لكن هذا الأمر يحتاج اتصال:

```
$object = Lingua::Translate->new(src => $tag_1, dest => $tag_2);
```

إذاً يمكنني أن أعرف هل يوجد اتصال أم لا من خلال هذا الأمر نفسه ولا داعي لإضافة المزيد من الأوامر للتأكد من وجود اتصال.

إن أُنشئ الكائن بنجاح فبالطبع هناك اتصال وإن لم ينشأ فهذا يعني أنه لا يوجد اتصال أو نوع آخر من الأخطاء، ولكن كما قلنا سابقاً أن الخطأ هذا لا يصلح أن نضعه في جملة شرطية if (object) لأنه في هذه الحالة لو كان الخطأ false سينهار البرنامج ولن يتابع الجملة الشرطية.

إذاً الحل أن نُقييم هذا الأمر وبناءً على النتيجة نأخذ التصرف المناسب ولكي لا تبدو الأسطر البرمجية القادمة غريبة فكروا بها على طريقة try في لغة جافا ولكننا هنا في بيرل نستخدم eval، وسنقوم بذلك كالتالي:

```
eval {
$object = Lingua::Translate->new(src => $tag_1, dest => $tag_2)
or die "Cannot Create an instance";
$message = $self->{origin_text}->GetValue();
$trans = $object->translate($message);
};

if ($@){$self->{translated_text}->SetValue("An Error occured: $@
\n Perhaps your not online.");
}

else{
$self->{translated_text}->SetValue($trans);
}
```

شرح الكود، في داخل eval نضع الأوامر التي قد ترجع أخطاء مميتة للبرنامج وهي أمر إنشاء الكائن والأمر الذي يوجد فيه method translate لأنه معتمد على أمر الكائن.

بعد الدالة نتأكد هل حصل خطأ؟ إن كان هناك خطأ نرجع رسالة الخطأ إلى المستخدم. ولكن سيبقى البرنامج يعمل بشكل طبيعي (هذه الحالة تشبه حالة المتصفحات مثل فايرفوكس فإذا لم يكن هناك اتصال بالإنترنت سيذهب البرنامج إلى حالة Offline وينتظر من المستخدم محاولة إعادة الاتصال)، أما إذ لم يوجد خطأ فنتابع البرنامج كما المعتاد ونرجع النص المترجم.

٨. مرحلة التحزيم.

هذه المرحلة النهائية في تطوير البرنامج حيث بعد الانتهاء من التخطيط والبرمجة والتصحيح وبعد التأكد من أن البرنامج وصل مرحلة نضج مناسبة يمكننا أن نقوم بتحزيمه Packaging والقيام بنشره.

طبعاً أساليب التحزيم تختلف كثيرا جداً بحسب اللغة ونظام التشغيل ونوع البرنامج. ولكن الهدف من هذه المرحلة هو توفير آلية سهلة للمستخدمين النهائيين كي يجربوا البرنامج فلا يعقل مثلاً أن أوزع برنامجي على أصدقائي بشكل كود C وأقول لهم عليكم بنائه (أي عمل Compile له) من المصدر! فالمستخدم النهائي غالباً لا يعرف كيف يقوم بإنشاء ملف تنفيذي من ملف المصدر، والحقيقة أن هذه ليست مشكلة المستخدم النهائي بل حتى المستخدمين المتقدمين في الأغلب سيحتاجون بعض الوقت والبحث كي يستطيعوا تشغيل برامج مكتوبة بلغة لم يتعاملوا معها من قبل فاللغات كثيرة جداً.

أسهل البرامج هي البرامج المترجمة Compiled ويمكن استخدام برامج لتسهيل عملية التنصيب للمستخدم كما نراه في الويندوز ولكن يعيبها أن الملف التنفيذي المبني لنظام معين سيحتاج إلى إعادة بناء في حال الرغبة بتشغيله في نظام آخر.

هناك أيضاً برامج الويب ولها طريقة تشغيل مختلفة وفي هذه الحالة يجب أن يكون هناك توثيق جيد لطريقة التنصيب على الخادم ويفضل إنشاء سكربت يقوم بمهمة التنصيب بدلاً من أن نترك هذه المهمة للمستخدم فهذا السكربت يجب أن يهتم بإنشاء جداول قواعد البيانات وإسناد القيم المناسبة للاختيارات المناسبة حسب طلب المستخدم وإعطاء التصاريح المناسبة إلى آخره.

الخلاصة أياً يكن نوع البرنامج يجب أن نحاول قدر المستطاع أن نوفر آلية سهلة لتشغيل البرنامج بدلاً من ترك المستخدم في متاهة التشغيل والبحث عن المترجم أو المفسر المناسب وبعدها البحث عن المكتبات التي يحتاجها البرنامج، فيترك البرنامج لأنه لا يستحق العناء.

التطبيق: في حالتنا استخدمنا لغة مفسرة وهي بيرل (أمثلة أخرى هي بايثون وروبي وغيرهم) والتي تعطينا ميزة جيدة وهي الانتقالية بين أنظمة التشغيل المختلفة ولكن يعيبها هو وجوب وجود المفسر على جهاز المستخدم. إذاً سيكون هناك خيارين بالنسبة لبرنامج بيرل (طبعاً باستثناء خيار توفير المصدر فقط):

- **الأول:** أن نرفق المفسر (حجمه صغير نسبياً) مع برنامجنا فيقوم المستخدم بتنصيب المفسر ثم تشغيل البرنامج.
- **الثاني:** أن نستخدم PAR أو PerlApp أو Perl2exe وهذه كلها أدوات لإنشاء ملف تنفيذي جاهز لبرامج بيرل. وبهذا لن يحتاج المستخدم إلى تنصيب أي شيء فقط ضغطتين على البرنامج وسيعمل (ملاحظة هامة: هذه الأدوات في حقيقة الأمر لا تقوم بعملية Compile كل ما في الأمر أنها تحلل البرنامج وتستورد الأجزاء المطلوبة فقط من مفسر بيرل والمكتبات اللازمة وتحزمها مع بعض).

ملاحظات بخصوص البرنامج

أولاً: في السنوات الأخيرة وجدنا توجه كبير إلى نقل البرامج من سطح المكتب إلى الويب والموبايل ولكن أيضاً هناك توجه إلى نقل برامج الويب إلى سطح المكتب فأغلب المدونين في تويتر مثلاً لا يدخلون إلى موقع تويتر وإنما يقومون باستخدام برامج سطح مكتب للقيام بهذه المهمة وهناك الكثير من هكذا برامج أو ما يسمى بالعميل.

في أغلب الأحيان الهدف منها هو إضافة ميزات جديدة إلى الخدمة الأصلية أو الاستفادة من قدرات الجهاز بشكل أفضل ولكن هناك أيضاً سبب آخر وهو أن البعض يفضل استخدام برامج سطح المكتب بدلاً من تكرار عمليات الدخول والخروج إلى المتصفح لذلك هناك رواج كبير لبرامج الرفع المباشر إلى فلكر مثلاً وغيرها من الخدمات.

ثانياً: في هذا الموضوع قمت بتمثيل بناء البرنامج على شكل برنامج سطح مكتب ولكن كان بالإمكانيات أن نستهدف منصات أخرى مثلاً أن نجعل البرنامج خاص للهواتف المحمولة كنظام أندرويد وآيفون وغيرهم بهذه الفكرة.

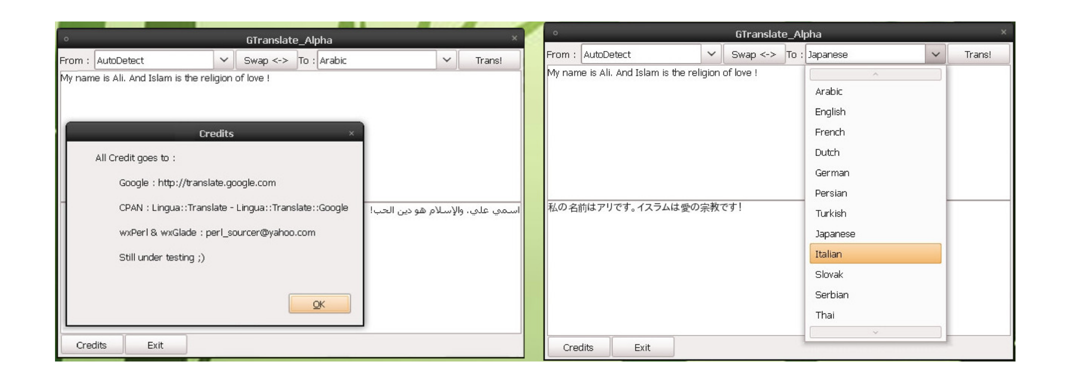
أيضاً كان بالإمكان أن نستهدف جعل البرنامج يعمل كإضافة لمتصفح كفايرفوكس أو كودجت Widget لسطح المكتب. بل حتى كان يمكن أن نُضمّن البرنامج في برنامج أكبر مثلاً لو كان لدينا برنامج ويب صممناه ليعمل كمجلة أو مدير مقالات كان بالإمكانيات أن نضمن هذه الميزة في البرنامج. فعلينا البحث عن المنصات التي سيلاقي فيها البرنامج شعبية أكثر.

ثالثاً: طبعاً كان بالإمكان إضافة المزيد من الخيارات لهذا البرنامج مثل ترجمة موقع أو ترجمة ملف نصي موجود في الجهاز إلخ، ولكن هذا ليس هدف الموضوع.

خامساً: تعمدت إلغاء مرحلة كتابة الاختبارات Tests لأنها طويلة وخاصة جداً بلغة بيرل وستختلف كلياً مع أي لغة أخرى فلن تقع في فائدة الأعضاء كثيراً خصوصا أنني أريد أكون عاماً بأكبر شكل ممكن.

ملاحظات عامة

تكلمنا عن بعض المراحل المهمة في تطوير البرامج، ولكن هناك الكثير من النقاط المهمة التي يجب الانتباه لها عندما يكون Our App In Action!



أولاً: لا تُعد اختراع العجلة! من الأشياء المهمة التي يجب أن ننتبه لها والتي قمنا بتطبيقها في هذا الموضوع هو عدم تكرار الجهود، فهذا البرنامج مثلاً كان بالإمكانيات أن أطور له قاعدة كلمات بنفسي ولكن هذا شيء مكلف جداً بالنسبة لشخص واحد وفي النهاية سأترك الإضافة إلى قاعدة بيانات الكلمات عندما أدرك أن هذا الأمر فوق طاقتي. فهذه المهمة تحتاج إلى متطوعين ومساهمين الأمر الذي قد لا يتوفر لي.

وفي الجهة المقابلة بدلاً من استخدام المكتبة الجاهزة التي استخدمتها للاتصال بجوجل كان أيضاً بإمكانياتي أن أقوم بعملية الاتصال بنفسي وذلك باستخدام LWP (World-Wide Web library for Perl) ولكن هذا سيحتاج إلى مزيد جهد وعناء وفي أغلب الأحيان لن تكون نتيجة أكوادي أفضل من المبرمجين الذين قاموا بإنشاء المكتبة التي استخدمناها لأنها طُورت من قِبَل مجموعة من الأشخاص المحترفين واختبرت بشكل مكثف من أشخاص آخرين. ولو على فرض كانت أكوادي أفضل من أكواد هذه المكتبة فالعناء الذي سأتحصل عليه أكبر من الفائدة العائدة.

ثانياً: تقسيم البرنامج إلى وحدات! في هذا الموضوع مجموع الأسطر البرمجية كانت ٢٢٤ سطر برمجي ولكن لاحظنا سهولة البحث عن الأخطاء والتعديل على أي شيء في البرنامج بسهولة تامة (و سيظل الأمر هكذا حتى لو ضاعفنا عدد هذه الأسطر مرات ومرات) وهذا راجع لكون البرنامج مقسم إلى وحدات منطقية تسهل عملية البحث والتصحيح وتتبع الأخطاء بدلاً من أن يكون البرنامج كله وحدة واحدة فتصعب عملية تتبع مصدر الأخطاء ومعرفة ماذا يؤثر على ماذا!

وسنذكر بعض أهم الأمور الأساسية التي تساعد على جعل البرنامج قابلاً للتطوير والتوسع Scalable والتعديل Maintainable:

١. تقسيم البرنامج إلى وحدات منطقية.
٢. تقليل اعتمادية كل وحدة على وحدات أخرى Dependency.
٣. عدم جعل المتغيرات مشاعة Global بل جعلها محصورة في أضيق مجال ممكن بحيث لا يمكن تغيرها من خارج مداها ولاحظنا هذا طوال البرمجة التي قمنا بها في هذا الموضوع ولم نستخدم متغير عام إلا مرة واحدة وذلك بهدف الإعلان المبكر.
٤. استخدام تسميات واضحة للمتغيرات والدوال والكلاسات.
٥. كتابة الملاحظات على الأجزاء التي فيها صعوبة أو فكرة.

سأكتفي بهذا القدر، ولكن سأختم بملاحظات بسيطة وهي في طلب المساعدة في المنتديات العربية أو الأجنبية:

١. تجنب الطلبات العامة! مثلاً كيف أبرمج موقع ويب؟ أو كيف استخدم المكتبة الفلانية؟ بل الأفضل تحديد السؤال في أمر أو دالة معينة.

٢. ابحث قبل السؤال.

٣. المحاولة قبل السؤال! فالتفاعل مع السائل يزيد كثيراً عندما يرون أنه بحث وسعى بنفسه ثم عرض مشكلته.

٤. اعرض الكود البرمجي! وهذه كثيرا ما تتكرر فالكثير لا يقبل بعرض كامل الكود المصدري، ربما هو خوف الإحراج أو خوف أن يسرق الكود أو شيء من هذا القبيل. ولكن عرض الكود البرمجي كاملاً وليس جزءً منه يساعد كثيراً على حل المشكلة.

٥. اجعل سؤالك طلباً وليس أمراً!

٦. وفي النهاية لا عيب من السؤال في طلب العلم.